PERQ
Systems
Corporation

PERQ MICROPROGRAMMER'S

REFERENCE MANUAL

March 1984

This manual is for use with POS Release G.5 and subsequent releases until further notice.

This document is not to be reproduced in any form nor transmitted in whole or in part, without the prior written authorization of PERQ Systems Corporation.

The information in this document is subject to change without notice and should not be construed as a commitment by PERQ Systems Corporation. The company assumes no responsibility for any errors that may appear in this document.

PERQ Systems Corporation will make every effort to keep customers apprised of all documentation changes as quickly as possible. The Reader's Comments card is distrib&uted with this document to request users' critical evalu&ation to assist us in preparing future documentation.

PERQ and PERQ2 are trademarks of PERQ Systems Corporation.

PREFACE

This document provides enough information to allow a microprogrammer to successfully use and exploit the features of the PERQ1 and PERQ1A microprogrammable CPUs. The document assumes that the reader is unfamiliar with PERQ but has some prior experience with horizontally programmed microEngines.

The PERQ1A CPU is an enhanced version of the PERQ1 CPU. The PERQ1A CPU is optional on the PERQ and standard on the PERQ2. Some references to the PERQ1A call the CPU the 16K ControlStore CPU. The design of the enhancements was made with certain goals in mind:

Current Microcode should run essentially unmodified;

Board space is at a premium and must be conserved;

Timing in certain critical chains must not be degraded.

The following features are available in the PERQ1A CPU but not in the PERQ CPU:

16K writable control store.

A 14 bit computable Goto with the address coming from the processor shift output.

Single precision multiply step and divide step hardware.

A base register for addressing the X and Y registers.

A readable victim latch.

Ability to use a long constant in a microinstruction which pushes the EStk.

This document is intended to be self-contained. A goal of this document is to explicitly identify implementation-dependent behavior as such. The document provides a full and complete description of the PERQ microprogramming language.

CHAPTER 1

INTRODUCTION

The syntax is described with a meta-language called EBNF (extended BNF). The following meta-symbols are used.

' '		– surround literal text.
|		– separate alternatives.
[ ]		– surround optional parts.
{ }		– surround parts which may be repeated zero or more times.
( )		– are used for grouping.
.		– ends a description.

1.1 FORMAT

Type programs in free format; a single micro-instruction can extend to as many lines as desired. You can insert blank lines and lines consisting only of comments anywhere. The exception to this rule is that a new micro-instruction must begin with a new line; you cannot place more than one instruction per line.

1.2 NAMES

Names can be any length, but only 10 characters are significant when two names are compared.

1.3 COMMENTS

Indicate comments by an exclamation mark (!). The remainder of the line following the exclamation mark is ignored. Comments can also be enclosed in braces Pascal style: '{' and '}' or '(*' and '*)'.

1.4 CONSTANT EXPRESSIONS

Expressions are allowed in most places where numeric values are required. In a few instances, only numbers or named constants are allowed and expressions are not. The syntax of constant expressions mimics that of Pascal. Expressions consist of operators and operands with certain precedence rules. Parentheses are used for controlling the order of evaluation.

As in Pascal, operators fall into one of three precedence classes: Multiplying operators have the highest priority, adding operators are next, and relational operators have the lowest priority. The multiplying operators are:

*	Signed integer multiply
div	Signed integer divide
mod	Signed integer remainder
and	Bitwise logical product
nand	Inverted bitwise logical product
lsh	Left shift
rsh	Right shift
rot	Right rotate

Lsh, rsh, and rot shift their lefthand operands the number of bits specified by their righthand operands.

The adding operators are:

+	Signed integer sum
-	Signed integer difference
or	Bitwise logical sum
nor	Inverted bitwise logical sum

The relational operators are:

=	Signed integer equal-to
<>	Signed integer not-equal-to
<	Signed integer less-than
>	Signed integer greater-than
<=	Signed integer less-than-or-equal-to
>=	Signed integer greater-than-or-equal-to
xor	Exclusive-or
xnor	Inverted exclusive-or

Xor and xnor are considered to be relational operators because they perform bitwise equality and inequality operations. The integer comparison operators return 0 for false and 1 for true.

In addition to the three precedence classes, three unary operators are available:

```
+        Unary integer identity
-        Unary integer negation
not      Unary bitwise complement
```

The unary identity and negation fall between the adding operators and the multiplying operators in precedence. The unary complement is higher priority than the multiplying operators.

Constant expressions are computed using 16-bit arithmetic and no overflow checks are applied (with the exception of a check for division by zero). In order to eliminate certain syntactic ambiguities, constant expressions must often be surrounded by parentheses. For example, the expression

```
not 1
```

could be interpreted as an ALU expression or a constant expression. For example,

```
R := not 1;     ! complement 1 at execution time
R := (not 1);   ! complement 1 at assembly time
```

The first form is preferable for constants which are less than 255 because the assembler can use a short constant, whereas the second form requires a long constant. This can be used to your advantage to create a 16-bit value with a short constant:

```
R := not (not 177400);  ! mask 177400 is formed by
! the short constant 377
```

1.5 SYNTAX

    name = letter {letter | digit} .

    number = ['2#' | '8#' | '10#' | '#'] digit {digit} .

    constant = name | number .

    register = [''] name .
             | '' constant .
             | '' '(' ConstExpr ')' .

    label = name .

    empty = .

MicroProgram = {Instruction ';'} 'end' ';' .

ConstExpr = ConstSimpleExpr ConstRelOp ConstSimpleExpr .

   ConstRelOp =   '=' | '<>' | '<' | '>'
              | '<=' | '>=' | 'xor' | 'xnor'.

   ConstSimpleExpr = ['+' | '-'] ConstTerm {ConstAddOp ConstTerm}.

     ConstAddOp = '+' | '-' | 'or' | 'nor' .

     ConstTerm = ConstFactor {ConstMulOp ConstFactor} .

       ConstMulOp =   '*' | 'div' | 'mod' | 'and'
                  | 'nand' | 'lsh' | 'rsh' | 'rot' .

       ConstFactor = {'not'} (constant | '(' ConstExpr ')') .

Instruction = {label ':'} Phrase {',' Phrase} .

   Phrase = empty
          | Pseudo
          | PascalStyleConstant
          | {Result ':='} ALU
          | Jump
          | Special .

```
Pseudo =  'Define' '(' ['''] name ',' ConstExpr ')'
        | 'Constant' '(' name ',' ConstExpr ')'
        | 'Opcode' '(' [ConstExpr ','] ConstExpr ')'
        | 'Loc' '(' ConstExpr ')'
        | 'Binary'
        | 'Octal'
        | 'Decimal'
        | 'Nop'
        | 'Case' '(' ConstExpr ',' ConstExpr ')'
        | 'Place' '(' ConstExpr ',' ConstExpr ')' .
```

```
PascalStyleConstant = name '=' ConstExpr .
```

```
  Result = register
          | 'TOS' | 'MA' | 'MDO' | 'BPC'
                  | 'SrcRasterOp' | 'DstRasterOp'
                  | 'WidRasterOp' | 'MQ' | 'RBase" .
```

```
ALU = ['not'] (Amux | Bmux)
    | Amux Op ['not'] Bmux
    | Amux '+' Bmux ['+' 'OldCarry']
    | Amux '-' Bmux ['-' 'OldCarry']
    | Amux 'amux' Bmux
    | Amux 'bmux' Bmux
    | 'MQ'
    | 'Victim' .
```

```
  Amux = register | 'Shift' | 'NextOp' | 'IOD'
       | 'MDI' | 'MDX' | 'TOS'
       | 'UState' ['(' register ')'] .
```

```
  Bmux = register | constant | '(' ConstExpr ')' .
```

```
  Op = 'and' | 'or' | 'xor' | 'nand' | 'nor' | 'xnor' .
```

```
Jump = ['If' Condition] Directive ['(' Target ')'] .
```

```
  Condition =  'True' |  'False'  | 'BPC[3]' | 'C19'
            |  'IntrPend'         | 'Odd'    | 'ByteSign'
            |  'Eql' | 'Neq'      | 'Gtr'    | 'Geq'
            |  'Lss' | 'Leq' | 'Carry' | 'OverFlow'.
```

```
  Directive = 'Goto' | 'Call' | 'Return' | 'Next' | 'JumpZero'
            | 'LoadS' | 'GotoS' | 'CallS' | 'NextInst'
            | 'ReviveVictim' | 'PushLoad' | 'Vector' | 'Dispatch'
            | 'RepeatLoop' | 'Repeat' | 'JumpPop' | 'LeapPop'
            | 'Loop' | 'ThreeWayBranch' .
```

```
  Target = label | constant | 'Shift' .
```

```
Special = Nonary | Unary | Binary .
```

```
Nonary = 'WCSlow' | 'WCSmid' | 'WCShi' | 'LoadOp' | 'Hold'
       | 'StackReset' | 'Push' | 'Pop' | 'Fetch' | 'Fetch2'
       | 'Fetch4' | 'Fetch4R' | 'Store' | 'Store2'
       | 'Store4' | 'Store4R' | 'LatchMA' | 'ShiftOnR' .
       | 'MultiplyStep' | 'DivideStep' .

Unary = UnaryName '(' ConstExpr ')' .

  UnaryName = 'LeftShift' | 'RightShift' | 'Rotate' | 'IOB'
            | 'CntlRasterOp' .

Binary = BinaryName '(' ConstExpr ',' ConstExpr ')' .

  BinaryName = 'Field' .
```

1.6 NOTES ON THE SYNTAX

Numeric constants preceded by a '#' are octal constants.

Constants can be defined Pascal style:

    name = value;

This allows including a file which contains constant definitions into both a Pascal program and a micro-program.

The syntax allows constructions which are semantically incorrect. In other words, there are many combinations of actions which cannot be represented in a single instruction. For example,

```
TOS := MA := 10;              is valid, but

TOS := BPC := 10;             is invalid.
```

Section 2.3 shows which fields of a micro-instruction are used by a particular action. The rule is that a certain field may be used only once.   Thus since 'TOS :=' and 'BPC :=' both use the SF (special function) field, they both cannot be used in a single micro-instruction.

Some features of the hardware are specific to the PERQ1 or the PERQ1A. The assembler reflects this by restricting usage of these features. The percent sign which signals the use of the base register may only be used on the PERQ1A.   MQ, RBase, Victim, LeapPop, Goto(Shift), MultiplyStep, and DivideStep may only be used on the PERQ1A. LatchMA may only be used on the PERQ1.

Some goto types do not allow tests (are unconditional), and for some

the test is optional. Similarly, some do not allow addresses, some require them, and for some the address field is optional. For PERQlA, some goto types do not allow Shift to be used as the address. The following specifies the rules:

req - required, opt - optional, <blank> - not allowed

Goto type	test	address	Shift	
Goto	opt	req	opt	
Call	opt	req	opt	
Return	opt			
Next				
JumpZero				
LoadS		req	opt	
GotoS	opt	opt	opt	
CallS	opt	opt	opt	
NextInst		req		
ReviveVictim				
PushLoad	opt	req	opt	
Vector	opt	req		
Dispatch	opt	req		
RepeatLoop				
Repeat		req	opt	
JumpPop	opt	req	opt	
LeapPop	opt	req	opt	(PERQlA only)
Loop	opt			
ThreeWayBranch	opt	req		

1.7 PERQ HARDWARE ARCHITECTURE

PERQ is implemented with a high-speed microprogrammed processor which
has a 170 nanoseconds microcycle time.   The microinstruction is 48
bits wide.   Most of the data paths in the micro engine are 20 bits
wide.   The data coming in and out of the processor (for example, IO
and Memory data) are 16 bits wide.   The extra 4 bits allow the
microprogrammed processor to calculate real addresses in a 1 megaword
addressing space.   The assumption is that virtual addresses are kept
in a doubleword in memory but calculations on addresses can be single
precision within the processor. The programmer of the virtual machine
never sees the 20 bit paths.

The major data paths are diagrammed below:
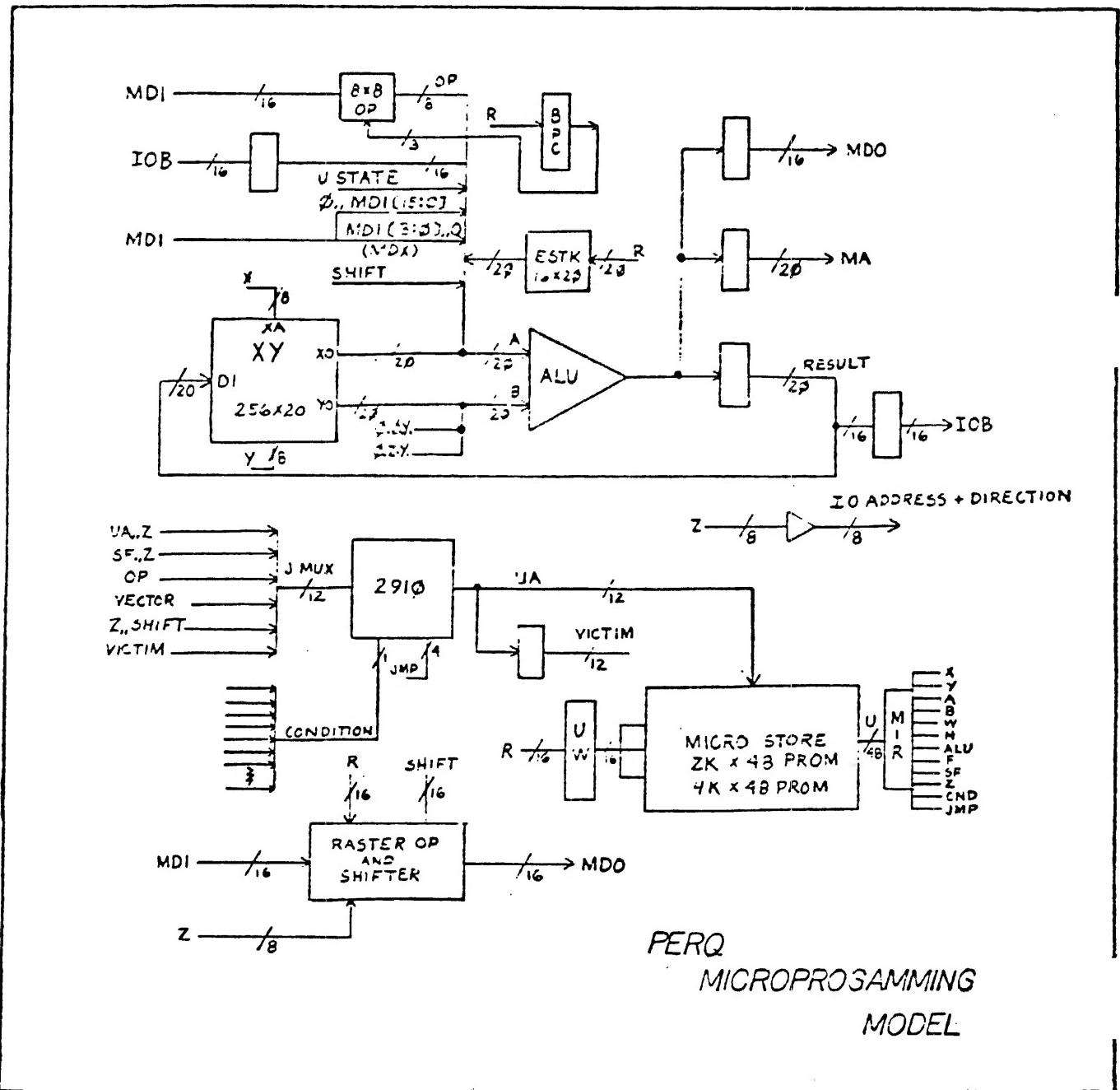
The XY registers (256 registers x 20 bits) form a dual ported file of general-purpose registers. The X port outputs are multiplexed with several other sources (the AMux) to form the A input to the ALU. The Y port outputs, multiplexed with an 8- or 16-bit constant via the BMux, form the B input to the ALU. The ALU outputs (R) are fed back to the XY registers as well as the memory data output and memory address registers. Memory data coming from the memory is sent to the ALU via the AMux. The IO bus (IOB) connects the CPU to IO devices. It consists of an 8-bit address (IOA) which is driven from a microword field and a 16-bit birectional data bus (IOD) which is read via AMux and written from R.

Opcodes and operands that are part of the instruction byte stream are buffered in a special 8 x 8 RAM (the Op file). The Op file is loaded 16 bits at a time from the memory data inputs. The output of the Op file is 8 bits wide and can be read via AMux or can be sent to the micro-addressing section for opcode dispatch. The read port of the Op file is addressed by the 3-bit BPC (Byte Program Counter).

A shift matrix (Shift), which is part of the special hardware provided for the RasterOp operator, can be accessed by loading an item to be shifted via the R bus, and reading the shifted result on AMux.

A 16-level push down stack (EStk) is written from R and read on AMux. The stack is used by the Q-code interpreter to evaluate expressions. BPC and the microstate condition codes can be read as the Micro State Register (UState) via AMux.

CHAPTER 2

MICROINSTRUCTIONS

Each microinstruction is a collection of 12 fields with a total of 48 bits, executed by the machine in 170 ns. Each of these 170 nanosecond intervals is refered to as a microcycle.

An "instruction" is a sequence of "Phrases". The microassembler produces microinstructions from one or more of these instructions. Each instruction can be associated with zero or more labels. Instructions are separated by semicolons (";").

NOTE: Some microinstructions, notably those produced by fetch and store type instructions, must be executed at specific microcycles, and influence several following microcyles.

The general syntax of a microinstruction is:

Instruction = {label ':'} Phrase {',' Phrase} .

```
    Phrase = empty
           ¦ Pseudo
           ¦ PascalStyleConstant
           ¦ {Result ':='} ALU
           ¦ Jump
           ¦ Special .
```


2.1 LABELS

A label is a name which can be prefixed to an instruction for use as a branching target. The syntax of a label is:

        label = name .

Instructions can have simple or compound labels, as described in sections 2.1.1 and 2.2.2.

2 - 1

2.1.1 Simple Labels

A simple label is a name, separated from its associated instruction
with a colon. This name can then be used as a target in a branch or
jump instruction.   A name is formed from a letter followed by any
number of letters or digits. The microassembler recognizes the first
ten letters or digits of a name. Names with more than ten letters or
digits, are only to the first ten letters or digits.

2.1.2 Compound Labels

Compound labels are formed by concatenating label-colon pairs to the
front of an instruction.   Thus, each instruction can have several
names. For example, the construction

        foobar:zeetix:looptarget: A + B + OldCarry;

allows the microinstruction 'A + B + OldCarry' to be named 'foobar',
'zeetix', or 'looptarget'.

2.2 PHRASES

A phrase is a syntactic building block, you use to build instructions.
An instruction is formed from a concatenation of phrases, with each
phrase separated by a comma (','). The assembler recognizes six types
of phrases. Chapter 3 describes the six types in detail. The syntax
is as follows:

    Phrase = empty
           ¦ Pseudo
           ¦ PascalStyleConstant
           ¦ {Result ':='} ALU
           ¦ Jump
           ¦ Special .

2.3 MICROINSTRUCTION FORMAT

Each 48-bit microinstruction is composed of 12 fields, as shown below.

```
    8        8       3 1 1 1   4    2    4       8      4     4
+---------+--------+---+-+-+-+-----+---+-----+---------+-----+-----+
¦   X     ¦   Y    ¦ A ¦B¦W¦H¦ ALU ¦ F ¦ SF  ¦    Z    ¦ CND ¦ JMP ¦
+---------+--------+---+-+-+-+-----+---+-----+---------+-----+-----+
```
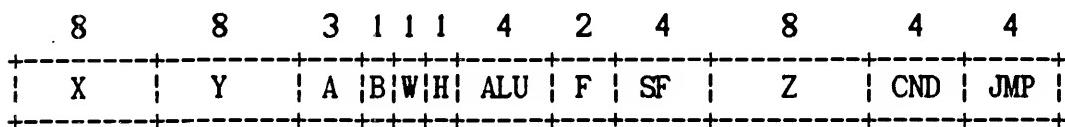
If the base register is set to zero, or is not loaded after booting,
register addressing on the PERQ1A is compatible with the PERQ1. In
this case, the $NOBASE assembler option may be used so that the
assembler will not require the percent sign prefix.

A brief description of each field is presented in sections 2.3.1 through 2.3.12. The microassembler produces microinstructions based upon the instructions described in detail in Chapter 3.

2.3.1 X (bits 47..40)

The X-field contains the address for the X port of the XY register file. This same field is used to reference a given register in the XY file for a register write operation.

2.3.2 Y (bits 39..32)

The Y-field contains the address for the Y port of the XY register file. It can also be used as the low order byte of a constant.

2.3.3 A (bits 31..29)

The A-Field contains the select lines used to drive the Amux. The A-field is encoded as follows:

A Field	Selects
0	Shifter output
1	NextOp (Opfile[BPC])
2	IOD (IO Data bus)
3	MDI (Memory Data inputs). See section 3.3.2.1 for more details.
4	MDX (Memory Data input, extended). See section 3.3.2.1 for more details.
5	Microstate register (UState).
6	XY register at location specified by the X field.
7	Expression Stack.

2.3.4 B (bit 28)

The B-field contains the Bmux select line. While B remains set, the Bmux selects a constant. While B remains cleared, Bmux selects the register specified by the Y-field contents.

2.3.5 W (bit 27)

The Write bit, while set, causes the contents of R to be written into
the register specified in the X field. While reset, no XY registers
are modified.

2.3.6 H (bit 26)

The Hold bit, while set, prevents IO devices from accessing memory.
It is also used with the JMP field (see section 2.3.12) to modify
address inputs.

2.3.7 ALU (bits 25..22)

The ALU field encodes the function used by the ALU to combine the A
and B inputs to the ALU. It is encoded as follows:

ALU Field	ALU Function
0	A
1	B
2	NOT A
3	NOT B
4	A AND B
5	A AND NOT B
6	A NAND B
7	A OR B
10	A OR NOT B
11	A NOR B
12	A XOR B
13	A XNOR B
14	A + B
15	A + B + OldCarry
16	A - B
17	A - B - OldCarry

OldCarry is the carry from the immediately preceeding microinstruction and is used for multiple precision arithmetic.

2.3.8 F (bits 21..20)

The Function field controls the interpretation of the SF and Z field contents. For PERQ1, it is encoded as follows:

Function	SF Use	Z use
0	Special Function	Constant/Short Jump
1	Memory Control	Short Jump
2	Special Function	Shift Control
3	Long Jump	Long Jump

For PERQ1A, it is encoded as folows:

Function	SF Use	Z use
0	Special Function	Constant/Short Jump
1	Memory Control and extended Special Func.	Short Jump
2	Special Func.	Shift Control
3	Long Jump	Long Jump

2.3.9 SF (bits 19..16)

While the function field (see 2.3.8 above) selects Memory Control, the Special Function bits contain the memory control bits. While the function field selects Long Jump, the Special Function field contains the four high-order memory address bits. Otherwise, the Special Function field selects the following functions:

SF	FUNCTION
0	LongConstant
1	ShiftOnR
2	StackReset (clear the EStk)
3	TOS := (R) (Top of EStk)
4	Push (the Estack)
5	Pop (the Estack)
6	CntlRasterOp := (Z)

```
7       SrcRasterOp := (R)
10      DstRasterOp := (R)
11      WidthRasterOp := (R)
12      LoadOp (OP := MDI)
13      BPC := (R)
14      WCS[15..0] := (R)
15      WCS[31..16] := (R)
16      WCS[47..32] := (R)
17      IOB Function
```

When used as Memory Control:

```
10      Fetch4R   Fetch 4 words in reverse order
11      Store4R   Store 4 words in reverse order
12      Fetch4    Fetch 4 words
13      Store4    Store 4 words
14      Fetch2    Fetch 2 words
15      Store2    Store 2 words
16      Fetch     Fetch 1 word from memory
17      Store     Store 1 word from memory
```

For PERQ1A when used as Extended Special Function:

```
0       (R) := Victim Latch
1       Multiply step or divide step
2       Load multiplier or dividend
3       Load base register
4       (R) := product or quotient
5       Push long constant (EStk)
6       2910 address inputs := Shift
7       Leap address generation
```

The following example performs a single precision divide of a single
precision dividend and a single precision divisor yielding a single
precision quotient and a single precision remainder.

```
     Constant(OffMultiply, 0);
     Constant(OffDivide, 0);
     Constant(UnSignedDivide, 100);
     Constant(UnSignedMultiply, 200);
     Constant(SignedMultiply, 300);

     Define(Dividend, 200);
     Define(Divisor, 202);
     Define(Quotient, 203);
     Define(QuotientSign, 204);
     Define(RemainderSign, 205);
```

```
      Tos := 0, Push;                    ! 0 for two's complementing
      RemainderSign := DividendHigh, RightShift(0); ! set sign of Mod
      QuotientSign := Shift xor Divisor, ! set sign of Div
                                         if Geq Goto(A);
      DividendLow := Tos - DividendLow;! abs value of dividend
      DividendHigh := Tos - DividendHigh - OldCarry;
      Divisor;
   A: if Gtr Goto(B);                    - ! if divisor >= 0
      Divisor := Tos - Divisor;          ! abs value of divisor
   B: Rotate(10#15);                     ! shifter must rotate left 1
      MQ := Dividend;                    ! load dividend
      WidRasterOp := UnSignedDivide;     ! set unsigned divide
      LoadS(10#15);                      ! S := 10#15
      Remainder := 0,                    ! initialize remainder
                DivideStep;              ! get started
   C: Remainder := Shift - Divisor, DivideStep, Repeat(C);! 10#16 steps
      WidRasterOp := OffDivide,          ! turn off divide hardware
                            if Geq Goto(D); ! if remainder >= 0
      Remainder := Remainder + Divisor;! correct remainder
   D: Quotient := MQ;                    ! read quotient
      QuotientSign;
      RemainderSign, if Geq Goto(F);     ! if quotient should be >= 0
      Quotient := Tos - Quotient;        ! set negative quotient
      RemainderSign;
   F: if Geq Goto(Done);                 ! if remainder should be >= 0
      Remainder := Tos - Remainder;      ! set negative remainder
   Done: Pop;                            ! restore stack
```

2.3.10 Z (bits 15..8)

The Z-field contains the low eight bits of a jump address, the high
eight bits of a Constant, shift control, or an IOB address, depending
on the state of the Function (F) and Special Function (SF) fields.

The encodings of the F field do not necessarily enforce restrictions
on the use of the Z field, they merely enable some of them. In
particular, B = 1, SF = 0, and F = 0 or 2 selects a long constant
using the Z field. For PERQ1A, F = 1 and SF = 5 also selects a long
constant using the Z field. In the PERQ1, long constants and special
functions may not be used in the same microinstruction. The PERQ1A
has a special function which pushes the EStk and selects a long
constant in the same instruction. The programmer need not select this
special function explicitly--the assembler takes care of it. When F
<> 2, the Z field is used for a jump address. When SF = 17 and F = 0
or 2, the Z field is used for an IOB address. When F = 2, the Z field
is loaded into the Shift Control register. These are the only
specific actions taken by the hardware that affect the usage of the Z
field. The hardware does nothing to prevent the Z field from being
used for several things at once. For example, it could be used for a
long constant and a jump address at the same time, or it could be used
as an IO address and a jump address at the same time. The assembler,

however, will flag an error if the programmer tries to load two different values into the same microinstruction field.

2.3.11 CND (bits 7..4)

The Condition (CND) field determines what to test during a conditional jump. They are encoded as follows:

CND	Test
0	True - always jump
1	False - never jump
2	IntrPend - interrupts pending
3	Spare (unused)
4	BPC[3] - Op File is empty
5	C19 - no carry out of bit 19 of the ALU (R[19])
6	Odd - ALU bit 0 (R[0])
7	ByteSign - ALU bit 7 (R[7])
10	Neq - Not equal to
11	Leq - Less than or equal to
12	Lss - Less than
13	Overflow - 16 Bit overflow in the ALU
14	Carry - carry out of bit 15 of the ALU (R[15])
15	Eql - Equal to
16	Gtr - Greater than
17	Geq - Greater than or equal to

2.3.12 JMP (bits 3..0)

The JMP field is described in detail in the AMD 2910 documentation; phrases which use it are described in detail in section 3.4.

For PERQ1, the JMP field is encoded as follows:

CIA	=	Current Instruction Address.
NIA	=	Next Instruction Address.
Addr	=	CIA[11:8],,Z[7:0] (Short)
		or SF[3:0],,Z[7:0] (Long).
S	=	Internal Address Register.
CStk	=	Top of 5-Level Call Stack.
OpCode	=	Z[7:6],,(not OpFile[BPC])[7:0],,Z[1:0].
Vector	=	Z[7:2],,0,,(not Device)[2:0],,Z[1:0].
Dispatch	=	Z[7:2],,(not Shift)[3:0],,Z[1:0].
Push	=	Push CIA+1 onto call stack
Pop	=	Pop call stack

CODE	NAME	PASS	FAIL
0	JumpZero	NIA := 0 Clear CStk	NIA := 0 Clear CStk
1	Call	NIA := Addr Push CStk	NIA := CIA + 1
2	NextInst (H=0)	NIA := OpCode	NIA := OpCode
	ReviveVictim (H=1)	NIA := Victim	NIA := Victim
3	Goto	NIA := Addr	NIA := CIA + 1
4	PushLoad	NIA := CIA + 1 Push CStk S := Addr	NIA := CIA + 1 Push Cstk
5	CallS	NIA := Addr Push CStk	NIA := S Push CStk
6	Vector (H=0)	NIA := Vector	NIA := CIA + 1
	Dispatch (H=1)	NIA := Dispatch	NIA := CIA + 1
7	GotoS	NIA := Addr	NIA := S
10	RepeatLoop S <> 0	NIA := CStk S := S - 1	NIA := CStk S := S - 1
	S = 0	NIA := CIA + 1 Pop CStk	NIA := CIA + 1 Pop CStk
11	Repeat S <> 0	NIA := Addr S := S - 1	NIA := Addr S := S - 1
	S = 0	NIA := CIA + 1	NIA := CIA + 1
12	Return	NIA := CStk⁻ Pop CStk	NIA := CIA + 1

13	JumpPop	NIA := Addr Pop CStk	NIA := CIA + 1
14	LoadS	NIA := CIA + 1 S := Addr	NIA := CIA + 1 S := Addr
15	Loop	NIA := CIA + 1 Pop CStk	NIA := CStk
16	Next	NIA := CIA + 1	NIA := CIA + 1
17	ThreeWayBranch		
	S ◇ 0	NIA := CIA + 1 Pop CStk S := S - 1	NIA := CStk S := S - 1
	S = 0	NIA := CIA + 1 Pop CStk	NIA := Addr Pop CStk

For PERQ1A, the JMP field is encoded as follows:

```
CIA       =Current Instruction Address.
NIA       =Next Instruction Address.
CBank     =Current 4K microstore Bank.
Addr      =CBank[1:0],,CIA[11:8],,Z[7:0] (Short) or
           CBank[1:0],,SF[3:0],,Z[7:0]   (Long)   or
           Y[5:0],,Z[7:0]                (Leap).
S         =Internal Register/Count~er.
CStk      =Top of 5-level call stack.
OpCode    =Z[7:6],,(not OpFile[Bpc])[7:0],,Z[1:0].
Vector    =Z[7:2],,0,,(not device)[2:0],,Z[1:0].
Dispatch  =Z[7:2],,(not Shift)[3:0],,Z[1:0].
Push      =Push CIA + 1 onto call stack.
Pop       =Pop call stack.
(lo)      =Bits [11:0].
(hi)      =Bits [13:12].
```

CODE	NAME	PASS	FAIL
0	JumpZero	NIA := 0 Clear CStk	NIA := 0 Clear CStk
1	Call	NIA:=Addr Push CStk	NIA:=CIA+1
2	NextInst (H=0)	NIA(lo):=OpCode NIA(hi):=CBank	NIA(lo):=OpCode NIA(hi):=CBank
	ReviveVictim (H=1)	NIA(lo):=Victim NIA(hi):=CBank	NIA(hi):=Victim NIA(lo):=CBank

3	Goto	NIA:=Addr	NIA(lo):=CIA+1 NIA(hi):=CBank
4	PushLoad	NIA(lo):=CIA+1 NIA(hi):=CBank Push CStk(lo) S:=Addr	NIA(lo):=CIA+1 NIA(hi):=CBank Push CStk(lo)
5	CallS	NIA:=Addr Push CStk	NIA:=S Push CStk
6	Vector (H=0)	NIA(lo):=Vector NIA(hi):=CBank	NIA:=CIA+1 NIA(hi):=CBank
	Dispatch (H=1)	NIA(lo):=Dispatch NIA(hi):=CBank	NIA:=CIA+1 NIA(hi):=CBank
7	GotoS	NIA:=Addr	NIA:=S
10	RepeatLoop if S(lo) <> 0	NIA(lo):=CStk NIA(hi):=CBank S(lo):=S(lo)-1	NIA(lo):=CStk NIA(hi):=CBank S(lo):=S(lo)-1
	if S(lo) = 0	NIA(lo):=CIA+1 NIA(hi):=CBank Pop CStk(lo)	NIA(lo):=CIA+1 NIA(hi):=CBank Pop CStk(lo)
11	Repeat if S(lo) <> 0	NIA(lo):=Addr NIA(hi):=CBank S(lo):=S(lo)-1	NIA(lo):=Addr NIA(hi):=CBank S(lo):=S(lo)-1
	if S(lo) = 0	NIA(lo):=CIA+1 NIA(hi):=CBank	NIA(lo):=CIA+1 NIA(hi):=CBank
12	Return	NIA:=CStk Pop CStk	NIA(lo):=CIA+1 NIA(hi):=CBank
13	JumpPop (H=0)	NIA:=Addr Pop CStk(lo)	NIA(lo):=CIA+1 NIA(hi):=CBank
	LeapPop (H=1)	NIA:=Addr Pop CStk	NIA(lo):=CIA+1 NIA(hi):=CBank
14	LoadS	NIA(lo):=CIA+1 NIA(hi):=CBank S:=Addr	NIA(lo):=CIA+1 NIA(hi):=CBank S:=Addr

15   Loop   NIA(lo):=CIA+1   NIA(lo):=CStk
                            NIA(hi):=CBank   NIA(hi):=CBank
                            Pop CStk(lo)

16   Next   NIA(lo):=CIA+1   NIA(lo):=CIA+1
                            NIA(hi):=CBank   NIA(hi):=CBank

17   ThreeWayBranch
        if S(lo) <> 0   · NIA(lo):=CIA+1   NIA(lo):=CStk
                            NIA(hi):=CBank   NIA(hi):=CBank
                            Pop CStk(lo)
                            S(lo):=S(lo)-1   S(lo):=S(lo)-1

        if S(lo) = 0   NIA(lo):=CIA+1   NIA(lo):=Addr
                            NIA(hi):=CBank   NIA(hi):=CBank
                            Pop CStk(lo)   Pop CStk(lo)

CHAPTER 3

PHRASES

This chapter describes each of the phrases recognized by the microassembler.

The general syntax of a phrase is as follows:

```
Phrase = empty
       | Pseudo
       | PascalStyleConstant
       | {Result ':='} ALU
       | Jump
       | Special .
```

3.1 PSEUDO PHRASES

A "Pseudo" is a phrase which is used by the microassembler to alter its own state. Its syntax is:

```
Pseudo = 'Define' '(' ['''] name ',' ConstExpr ')'
       | 'Constant' '(' name ',' ConstExpr ')'
       | 'Opcode' '(' [ConstExpr ','] ConstExpr ')'
       | 'Loc' '(' ConstExpr ')'
       | 'Binary'
       | 'Octal'
       | 'Decimal'
       | 'Nop'
       | 'Case' '(' ConstExpr ',' ConstExpr ')'
       | 'Place' '(' ConstExpr ',' ConstExpr ')' .
```

Sections 3.1.1 through 3.1.10 describe the defined PSEUDOs.

3.1.1 'Define' '(' ['''] name ',' ConstExpr ')'

Associates a name with a certain register number. The ConstExpr must be in the range 0..255.

For example, the instruction

    Define(aRegister, #10);

informs the assembler that the name aRegister refers to register
number #10.   When the $PERQ1A and the $Base assembler options are
used, the percent sign () is required for registers with numbers less
than #100.

3.1.2 Constant

Associates a name with a numeric constant. For example,

    Constant(Fourteen, 2*7);

informs the assembler that the name Fourteen should be synonymous with
the constant 14.

3.1.3 Opcode

Assembles the current instruction into the location specified by the
following formula:

    Opcode(Op)   =>   (Op xor #377) lsh 2
    Opcode(Base, Op)   =>   (Op xor #377) lsh 2 + Base

For example, the instruction

    Opcode(1), Tos := Tos + 1;

assembles into the location

    (1 xor #377) lsh 2 = #376 lsh 2 = 1770

This computation matches the hardware for the NextInst jump so that
statements containing

    Opcode(JumpTableBase, OpcodeNumber), ...

can be used in conjunction with a jump of the form

    NextInst(JumpTableBase)

3.1.4 Loc

Assembles the current instruction into the location specified by
ConstExpr.

3.1.5 Binary

Make 2 the default base for numeric constants.

3.1.6 Octal

Make 8 the default base for numeric constants.

3.1.7 Decimal

Make 10 the default base for numeric constants.

3.1.8 Nop

A   placeholder.   Assembles   a   No-Op   instruction,   used   for
synchronization between the processor and the memory/IO systems.

In response to a NOP, the microassembler builds a MicroInstruction
with the X, Y, B, W, H, ALU, and CN fields equal to zero, and the A
field equal to 6 (select XY[X]). The JMP, SF, and Z fields are used
to encode a jump to the following instruction (in the source code).

> NOTE: The condition codes are not preserved during a Nop
> instruction because the hardware executes an ALU operation
> during every instruction, including Nop.

3.1.9 Case

Assembles the next instruction into the location specified by the
following formula:

```
Case(Val)        ==>    (Val xor #17) lsh 2
Case(Base, Val)  ==>    (Val xor #17) lsh 2 + Base
```

For example, the instruction

```
Case(#100, 1), Tos := Tos + 1;
```

assembles into the location

```
(1 xor #17) lsh 2 + #100 = #16 lsh 2 + #100 = #170
```

This computation matches the hardware for the Dispatch   jump   so   that
statements containing

    Case(JumpTableBase, CaseNumber), ...

can be used in conjunction with a jump of the form

    Dispatch(JumpTableBase)

3.1.10 Place

Makes a range of locations available for assembly. The statement

    Place(A, B)

specifies the range A to B (inclusive). Several Place statements can
be used, in which case the union of all   ranges   can   be   used.   This
function is used by the PLACER.

"Place" allows the microstore to be partitioned, so that explicit
subranges of it can be loaded without risk of   inadvertant   damage   to
the   remainder.   For   example,   a   range   of   the microstore could be
defined, using place, as an overlay area.   Attempts   to   load
microinstructions   outside   the   range specified in the place (through
the 'LOC' directive, for example) cause the microassembler to flag   an
error.

3.2 PASCAL STYLE CONSTANTS

The Pascal style constant definition

    name = ConstExpr

is functionally identical to

    Constant(name, ConstExpr).

This construction allows a constant definition file to be included by
both a Pascal program and a microprogram. The instruction causes all
references to the name to be interpreted as constants.

This is useful in defining entities such as register assignments or
parameters which must be used by both Pascal   programs   and   PERQ
MicroPrograms.

For example, suppose a file labeled "AFile" contained the following statements:

```
RegLoad    = 1002;
StartVal   = 1023;
StopVal    = 3049;
```

and a Pascal program and a Perq microProgram looked like:

PASCAL PROGRAM   PERQ MICROPROGRAM
...   ...
  Const $INCLUDE Afile
  $Include AFile   ...
  ...

The constants "RegLoad", "StartVal", and "StopVal" would be available to both the Pascal program and the MicroProgram, and a change to their value (in AFile) is conveniently passed into both.

The syntax for a Pascal Style Constant is:

    PascalStyleConstant = name '=' ConstExpr .

WARNING: The default base in Pascal is always Decimal, but the default base in microcode is usually Octal (but may be changed). Thus it is safest to use declarations of the form

    name = #number

which is interpreted as Octal in both Pascal and microcode. Alternatively, the microprogram could contain

```
Decimal;               ! change default base to 10
$Include AFile
Octal;                 ! change default base back to 8
```

3.3 {Result ':='} ALU

Sections 3.3.1 and 3.3.2 describe the primitives that form the constructions of section 3.3.3. Section 3.3.1 describes the optional result assignment locations and section 3.3.2 describes Amux, Bmux, the operators which drive the ALU, and the OldCarry bit. Section 3.3.3 describes the permissable constructions formable from Amux, Bmux, the operators, and the OldCarry bit.

The 20-Bit AMux output (AMUX[19:0]) and the 20-Bit BMux output (BMux[19:0]) form the A and B inputs to the ALU. The ALU output is latched in the R register (R[19:0]).

The syntax is:

```
Result = register
       | 'TOS' | 'MA' | 'MDO' | 'BPC'
             | 'SrcRasterOp' | 'DstRasterOp'
             | 'WidRasterOp' | 'MQ' | 'RBase" .
```

3.3.1 {Result ':='}

The ALU output can be written into zero or more RESULT locations, described in sections 3.3.1.1 through 3.3.1.8, below.

The assignment construction stores the full or partial result of the ALU (R[19:0]) in the designated target location. In addition to simple assignments (which store R in one location), compound assignments (which store R in several locations) can be constructed, subject to the constraint that each microinstruction field can only be assigned ONCE.

NOTE: The syntax allows multiple assignment of the same ALU output in the same microinstruction. Since each field of the microinstruction can only be assigned once (see section 2.3) per micro- instruction, care should be exercised in multiple assignments to prevent conflicts. The microassembler dissallows such conflicts.

Simple Assignments (result := ALU) copy the bits in R to the location specified by result, bit for bit. If the result location is less than 20 bits wide (BPC, for example), the assignment maps only those bits of R which also appear in result. For example,

BPC := IOD;

copies the four low-order bits of the IO data bus into the four-bit BPC register.

Compound Assignments (a := b := ... := ALU) copy the bits in R to each location specified so long as each result location can be specified without a conflicting use of any microinstruction field. For example,

DEFINE(Foo, #15), Foo := TOS := MA := 10, Fetch;

is valid (TOS uses the SF and F fields, MA uses no fields), while the instruction

TOS := BPC := 10;

is invalid (TOS uses the SF field in a way which conflicts with the BPC use of the same field). The microassembler dissallows these invalid combinations.

3 - 6

The syntax is

```
result  ::=   register
            | 'TOS'
            | 'MA'
            | 'MDO'
            | 'BPC'
            | 'SrcRasterOp'
            | 'DstRasterOp'
            | 'WidRasterOp'
```

3.3.1.1 Register

Directs the contents of R to Register. Register must be a previously defined register name (see section 3.1.1).

A base register facility is available on the PERQ1A.   Some suggested uses are:

1.) Saving and restoring register values in a loop.

2.) Establishing context dependent registers in a way that they may co-exist with a set of globally accessable registers.

3.) Using the registers as a deep stack.

The use of the base register is controlled by bits 6 and 7 of the X and Y fields of the microinstruction. For register numbers 0 through 77, the base register value is OR-ed with the X or Y field to form the register address. Register numbers 100 through 377 are not modified. Thus if the base register is loaded with 0, the PERQ1A register addressing mechanism is compatible with that of the PERQ1.

The base register is loaded from R with the "RBase := " special function. The value loaded into the base register is inverted when it is loaded. Thus to load a value V.into the base register, use "RBase := not V". The base register is cleared when the boot button is pressed. This is done to allow normal register access during boot sequences.

In order to help prevent inadvertant errors, the assembler requires the programmer to explicitly indicate whether a register is affected by the base register. This is required both at the point of definition and the point of use. Registers whose numbers are less than 100 must have a percent sign (%) prefixed to their names.

This instruction uses the X and W fields (X := address, W := 1).

3.3.1.2 TOS

A reference to the top of the expression stack. Each assignment to
TOS replaces the previous contents of the expression stack with the
current ALU output.

A reference to TOS uses the F and SF fields.

3.3.1.3 MA

The memory address register. MA is latched from R during fetch or
store type microinstructions. An assignment to the MA register does
not occupy any microinstruction fields, but is helpful as a mnemonic
aid.

NOTE: The MA register is latched from R during EACH fetch or
       Store type microinstruction. It is therefore a good
       programming practice to reflect this in an assignment to
       it, making its contents (for the fetch or store) explicit.

3.3.1.4 MDO

The memory data output register. Each assignment to MDO directs the
low-order 16 bits of the ALU output (R[15:0]) to MDO[15:0]. Like the
MA register, an assignment to MDO does not occupy any fields, but is
helpful as a mnemonic aid.

Note: The MDO register is latched from R during the data
       transfer portion of each store type instruction. It is
       therefore a good programming practice to reflect this in
       an assignment to it, making its contents explicit.

3.3.1.5 BPC

A reference to the four-bit BPC counter. An assignment to BPC loads
R[3:0] (the four low-order bits of R) into the BPC.

This instruction uses the F and SF fields (F:=0, SF:=8#13).

3.3.1.6 SrcRasterOp

SrcRasterOp is a control register in the raster-op/shifter hardware.
An assignment to SrcRasterOp loads R into the SrcRasterOp register.

This instruction uses the F and SF fields (F:=0, SF:=8#7).

3.3.1.7 DstRasterOp

DstRasterOp is a control register in the raster-op/shifter hardware.
An assignment to Dstrasterop loads R into the DstRasterOp register.

This instruction uses the F and SF fields (F:=0, SF:=8#10).

3.3.1.8 WidRasterOp

WidRasterOp is a control register in the raster-op/shifter hardware.
An assignment to WidRasterOp loads R into the WidRasterOp register.

This instruction uses the F and SF fields (F:=0, SF:=8#11).

3.3.1.9 RBase

The PERQ1A provide a base register facility. See section 3.3.1.1 for
a description of how the base register affects register addressing.
An assignment to RBase loads the complement of R[7:0] (the low order 8
bits) into the base register.

This instruction uses the F and SF fields (F:=1, SF:=3).

3.3.1.10 MQ

The PERQ1A provides hardware to assist integer multiply and divide.
See sections 3.5.1.16 and 3.5.1.17 for a description of the
multiply/divide hardware. As assignment to MQ loads R into the
multiplier/quotient register.

This instruction uses the F and SF fields (F:=1, SF:=2).

3.3.2 ALU

The syntax is

```
Amux   ::=  register
            | 'Shift'
            | 'NextOp'
            | 'IOD'
            | 'MDI'
            | 'MDX'
            | 'TOS'
            | 'UState' [ '(' register ')' ]

Bmux   ::=  register|constant

Op     ::=  'and'
            | 'or'
            | 'xor'
            | 'nand'
            | 'nor'
            | 'xnor'
```

3.3.2.1 AMUX

The microassembler uses the following primitives to control the AMUX.
The 20 AMux output bits (AMux[19..0]) form the A input to the ALU.

```
Amux = register | 'Shift' | 'NextOp' | 'IOD'
       | 'MDI' | 'MDX' | 'TOS'
       | 'UState' ['(' register ')'] .
```

Register directs the contents of the register addressed by the X field
to the AMux output. Register must be a previously defined register
name (see section 3.1.1). If part of an assignment construction, the
register specified in AMux be the same as the register being written.
For example, the statement:

    Foo := Foo + Bar;

works, while the statement

    Foo := Bar + Zee;

does not.

SHIFT directs the output of the shifter onto the AMux output lines.
This output is the shifted value of whatever was on R in the last
executed microinstruction.

NOTE: For meaningful results, the shift control function must
       have been specified during a prior microinstruction using

a LeftShift (section 3.5.2.1), RightShift (section 3.5.2.2), Rotate (section 3.5.2.3) or ShiftOnR (section 3.5.1.18) special function.

NEXTOP directs the contents of the OP File byte currently pointed to by the BPC onto the AMUX output lines, and BPC is incremented by one.

NOTE: If BPC overflows, the microinstruction eventually executes again. This precludes instructions such as "Foo := NextOp + Foo".

The microassembler automatically adds an "IF BPC[3] GOTO(Refill)" jump clause. Thus, if BPC overflows, control passes to refill. The instructions at 'refill' must increment UPC by four (pointing to the next quadword), set BPC to 0 (clearing the overflow), and start a Fetch4 to the Op File (loading the next quadword). The special function LoadOp must be executed in the t1 after the Fetch4 to cause the Op file to be loaded with the data coming in on MDI. Refill must then jump back to the instruction which needed the byte so that the instruction may be re-executed. This is easily accomplished with a 'ReviveVictim' jump directive (see section 3.4.2.10).

The PERQ1A can read the victim register directly; see Section 3.3.3.4.

WARNING: When a NextOp executes with BPC[3] set, the current address is loaded into the Victim register and locked. No new value is loaded until the Victim register is cleared by the ReviveVictim jump or by reading the Victim register directly.

IOD directs the contents of the IO databus to the 16 low bits of the AMUX output. Bits 19..16 are cleared.

MDI extracts a memory word; the contents of the Memory Data Bus are directed to the low-order 16 bits of the AMux output (AMux[15:0]). The remaining high order bits (AMux[19:16) of the output are cleared.

MDI is valid only during the four cycles which immediately follow a fetch-type instruction. Section 3.5.1 contains details of memory control timing.

MDX extracts a memory extension; the low order 4 bits of the Memory Data Bus (MD[3:0]) are directed to AMux[19:16]. Bits AMux[15:0] are cleared. MDX is used in conjunction with MDI to obtain a full 20 bit physical address.

For example, a 20 bit physical address at memory address FOO can be placed in register BAR as follows:

```
MA:= FOO, Fetch2; ! initiate the Fetch sequence
BAR := MDI;       ! get the first word
BAR := BAR Or MDX;! get the rest
```

TOS directs the contents of the top of the Expression stack to the AMux output.

USTATE [(Register_Name)] directs the contents of the microstate register to the AMux output. If Register_Name is specified, UState[15:12] contains the inverted high-order four bits of the selected register (using the Y field and BMux). If Register_Name is not specified, UState[15:12] contains Bmux[19:16] (inverted).

The UState register contains various interesting items packed in a single word. The UState register (A=5) looks like:

```
   19          16 15          12 11 10 9 8 7 6 5 4 3          0
   +------------+------------+--+--+--+--+--+--+--+--+----------+
   |     0      | BMux 19:16|uu|uu|SE|uu| N| C| Z| V|   BPC    |
   +------------+------------+--+--+--+--+--+--+--+--+----------+
```

uu	Unused
BPC	Byte Program Counter
N	Negative (ALU result < 0)
Z	Zero (ALU result = 0)
C	Carry (ALU carry out of bit 15)
V	Overflow (ALU overflow occured)
SE	EStk Empty (inverted data -- 0 = empty)
BMux 19:16	Upper 4 Bits of BMux, used to read bits 19:16 a register (inverted data).

NOTE: UState[15:12] contain the INVERTED register or BMux bits. They can be complemented again (using the NOT function, section 3.3.1) and a field qualifier can provide a mask. For example,

```
        NOT UState(aRegister), Field(14,4);
        aResult := shift;
```

3.3.2.2 BMUX

The microassembler uses the following primitives to control the BMUX. The 20 BMux output bits (BMux[19..0]) form the B input to the ALU.

The syntax is

Bmux ::= register | constant| '(' ConstExpr ')'

Register directs the contents of Register to the BMux output. The register name must have been previously defined (see section 3.1.1).

Constant directs a one or two byte constant onto the low order 8 or 16 bits of the BMux output (BMux[7:0] or BMux[15:0]). The high-order 4 bits of the BMux output (BMux[19:16]) are cleared. The microassembler determines the required length of the constant (one or two bytes) and

uses the F and SF fields accordingly.

Constant may be either implicit (through the use of a predefined constant name, section 3.1.2) or literal.

3.3.2.3 Operators

The microassembler recognizes several operators. These keywords are used by the microassembler to determine which ALU functions are to be enabled.

The syntax is

    Op = 'and' | 'or' | 'xor' | 'nand' | 'nor' | 'xnor' .

All of these instructions use the ALU field of the microinstruction.

AND - R gets the bitwise logical AND of AMux and BMux.

OR - R gets the bitwise logical OR of AMux and BMux.

XOR - R gets the bitwise logical XOR of AMux and BMux.

NAND - R gets the inverted bitwise logical AND of AMux and BMux.

NOR - R gets the inverted bitwise logical OR of AMux and BMux.

XNOR - R gets the inverted bitwise logical XOR of AMux and BMux.

3.3.2.4 OldCarry Bit

The OldCarry bit, used in several ALU constructions, contains the carry (or borrow) from bit 16 of the immediately preceding microinstruction, and is used to perform multiple precision arithmetic operations.

OldCarry contains a carry bit if the immediately preceding microinstruction was an addition and a borrow bit if the immediately preceding microinstruction was a substraction.

3.3.3 Constructions

This section presents valid ALU constructions. They are grouped into constructions which use one operand, constructions which imply logical operations between two operands, constructions which cause arithmetic operations among two operands (sometimes including the oldcarry bit for multiple precision arithmetic, and several special constructions used primarily for diagnostic purposes).

For each of these instructions, result, Amux, Bmux, the operator, and OldCarry is specified using the syntax defined in the previous sections.

Each of these constructions uses the ALU field of the microinstruction in addition to the fields used by the elements of the construction.

The syntax is

```
ALU = [ 'not'] (Amux | Bmux)
    | Amux Op [ 'not'] Bmux
    | Amux '+' Bmux [ '+' 'OldCarry']
    | Amux '-' Bmux [ '-' 'OldCarry']
    | Amux 'amux' Bmux
    | Amux 'bmux' Bmux
    | 'MQ'
    | 'Victim' .
```

3.3.3.1 Single Operand Constructions

Single operand instructions extract either AMux or BMux. They may be preceded by the keyword NOT, causing the bitwise inversion of the operand.

The syntax is

['not'] (Amux | Bmux)

AMux directs the AMux outputs to R[19:0]. When preceded by the keyword NOT, inverts the sense of R[19:0]. Amux is specified using one of the forms described in section 3.3.2.1.

BMux directs the BMux outputs to R[19:0]. When preceded by the keyword NOT, inverts the sense of R[19:0]. Bmux is specified using one of the forms described in section 3.3.2.2.

3.3.3.2 Double Operand logical Constructions

These constructions are used to form logical combinations of the Amux and Bmux outputs.

The syntax is

Amux Op ['not'] Bmux

AMux Operator BMux uses one of the operators described in section 3.3.2.2, and combines the Amux outputs with the Bmux outputs placing the result in R[19:0].

AMux Operator NOT BMux combines the Amux outputs (specified in section

3.3.2.1) with the INVERTED Bmux outputs (specified in section 3.3.2.2), placing the result in R[19:0].

NOTE: The following constructions are not implemented in the ALU:

Amux 'nand' 'not' Bmux
Amux 'nor' 'not' Bmux
Amux 'xor' 'not' Bmux
Amux 'xnor' 'not' Bmux

The microassembler disallows these four constructions.

3.3.3.3 Double Operand Arithmetic Constructions

These constructions are used to form arithmetic combinations of the Amux and Bmux outputs.

The syntax is

Amux '+' Bmux ['+' 'OldCarry']
|Amux '-' Bmux ['-' 'OldCarry']
 ...

AMux + BMux [+ OldCarry] forms the 20-bit binary sum of AMux and Bmux. The oldCarry bit is the carry from AMux[15] and BMux[15]. Thus, for multiple precision arithmetic, the appropriate sequence is:

1. Amux + Bmux ;   ! Low order word
2. Amux + Bmux + OldCarry; ! High-order word,

To minimize overhead during normal arithmetic operations from memory, the condition codes are set based upon the low-order 16 bits of the result [R[15:0]. Thus, while the four high-order bits of Amux and Bmux do participate in the addition, and the result is maintained in R[19:16], they DO NOT participate in the state of the condition codes. This affects the outcome of conditional branches (see section 3.4).

AMux - BMux [- OldCarry] forms the 20-bit binary difference of AMux and Bmux. The oldCarry bit is the borrow from AMux[15] and BMux[15]. Thus, for multiple precision arithmetic, the appropriate sequence is:

1. Amux - Bmux ;   ! Low order word
2. Amux - Bmux - OldCarry; ! High-order word,

To minimize overhead during normal arithmetic operations from memory, the condition codes are set based upon the low-order 16 bits of the result (R[15:0]). Thus, while the four high-order bits of Amux and Bmux do participate in the subtraction, and the result is maintained in R[19:16], they DO NOT participate in the state of the condition codes. This affects the outcome of conditional branches (see section

3.4).

The special operators AMUX and BMUX permit the state of both the AMUX outputs and the BMUX outputs to be specified while only one or the other is being directed to R[19:0].

In certain situations, primarily diagnostic, this is a necessary addition to the machine's functionality. For example, an ALU diagnostic can use these operators to identify Bmux bits which are inappropriately coupled to the ALU output, by setting both Amux and Bmux outputs to a known value and checking that the ALU generates the correct output.

The syntax is

          Amux 'amux' Bmux
        | Bmux 'bmux' Bmux

Amux AMUX Bmux causes the Amux outputs to be directed to R[19:0], while simultaneously putting a known value on the Bmux outputs.

Amux BMUX Bmux causes the Bmux outputs to be directed to R[19:0], while simultaneously putting a known value on the Amux outputs.

3.3.3.4 Special Constructions

The PERQ1A allows two more constructions to read the MQ (multiplier/quotient) and Victim registers. These are used as though they were ALU operations. For example

          Result := MQ;

Reads the MQ register and assigns its value to the register named Result. The Victim register (see section 3.3.2.1) is read in a similar way. Reading the Victim register clears it an makes it possible to load a new Victim value. The Victim register is defined only from the time a NextOp is executed with BPC[3] set until it is read or used in a ReviveVictim jump. Therefore the Victim register may only be read once.

WARNING: Reading MQ and Victim does not actually use the ALU and thus no computation may be done with the value. Reasonable actions are to assigne the value to a register or send it to the shifter. Note that since the value does not pass through the ALU, condition codes are invalid in the cycle which follows.

3.4 JUMP

Jump microinstructions are used to alter the sequential control flow of a microprogram. A Jump microinstruction is constructed from an

optional condition, a directive, and a target address.

A jump needing an address normally gets it from the Z field. Since Z is only eight bits wide, and the control store requires a 12 bit address, another four bits of address are needed.

The microassembler derives these other four bits based upon the target of the jump. Short jumps branch to a location on the same 256-word page as the current microinstruction (CIA). To go to an arbitrary location in the control store, the F field can specify a long jump (F=3), which uses the SF field for the upper four bits of address.

The PERQ1A has a writable controlstore that is expandable in multiples of 4K up to 16K instructions. The 4K multiples are referred to as banks of the controlstore. The microinstruction address paths are expanded to provide 2 more address bits. Hardware was added to provide 2910-like functions for the high order bits. This approach has numerous drawbacks because the 2910 is not an expandable bit slice. The problems are twofold. It is not possible to expand the 12 bit counter on the 2910 since it does not provide a carry out. Secondly, instructions such as 'RepeatLoop' and 'ThreeWayBranch' manipulate the control stack depending on the state of the counter. The state of the counter is not available external to the chip. This could lead to the two stacks getting 'out of sync' with each other.

The upper 2 and the lower 12 bits of the micro address are treated differently. In fact, the sequencer instructions will be different for the upper and lower bits. The jump control table shows the differences between them.

In addition to Short and Long jumps there is a level of address generation called Leap which is capable of addressing the entire 16K. A 2-bit bank register supplies the current bank address during Long and Short jumps. When we Leap, however, the 14-bit address will be formed from concatenation of the Y and Z fields, with the Y field supplying the most significant byte. The assembler generates leap addresses (by setting F = 1 and SF = 7) when the target address of a jump is in another bank. The assembler restricts a single assembly to one bank. This means leaps are required only for constant addresses in another bank.

The following notes are provided as an aid to reading the jump control table for PERQ1A.

1. Incrementing or decrementing of the microinstruction program counter and the S register do not affect the upper 2 bits. Thus these will not cross bank boundaries.

2. The low 12 bits of S are not available to the circuitry controlling the upper 2 bits of S and the upper 2 bits of the microinstruction program counter. Thus jumps which depend on whether S has reached zero will not cross bank

boundaries, and are allowed to push or pop only the lower
12 bits of the CStk.

3. Since Repeat, RepeatLoop, Loop, and ThreeWayBranch only pop
the lower 12 bits of the CStk, PushLoad only pushes the low
12 bits of the CStk. This means that the short version of
JumpPop must be used to exit such a loop.

4. Since Call and CallS push both parts of the CStk, the long
version of JumpPop (called LeapPop) must be used to clear
the call stack.

5. NextInst, Vector, and Dispatch jumps may not be used across
bank boundaries.

The micro-assembler provides minimal support for Leaping jumps. A
single assembly must fit entirely inside a 4K bank of the
controlstore. This means that Leaps are allowed only with constant
addresses or with the Goto(Shift) described in Section 3.4.3.3. The
assembler, therefore, always knows whether to generate a Leap.

Since the JumpPop type has two variants to control whether to pop the
upper stack, a new jump type is necessary: LeapPop. This is a JumpPop
which pops the upper stack.

For both PERQ1 and PERQ1A, the address for jumps might not come from
the Z, SF, or Y fields.

In addition to control-store addresses which come from the Z and SF
fields, a jump address can also come from the following sources:

SOURCE	DESCRIPTION
The S register	Internal to the microsequencer
The Call Stack	Five-level stack internal to the microsequencer
Current Address + 1	Uses full 12-bit Address
Victim Register	Hardware register, contains the address of the most recent 'failed' NextOp.
Processor Shifter	For PERQ1A.

There are three jumps which are multi-way branches. For each of these
three jumps, the Z field provides the address bits needed to complete
the 12-bit control-store effective address.

DIRECTIVE   DESCRIPTION

NextInst   256-way branch, based on the next byte from the Op
                file. The next byte provides bits 9..2 of the
                effective address (inverted); bits 11..10 come from
                Z[7:6] and bits 1..0 come from Z[1:0]. This results in
                a 256-way branch with a spacing of four instructions.

```
                               +--+--+--+--+--+--+--+--+
        OP File (next byte): | 7| 6| 5| 4| 3| 2| 1| 0|
                               +--+--+--+--+--+--+--+--+
            !                    !  !  !  !  !  !  !  !
            !                    !  !  !  !  !  !  !  !
           not                   !<---I N V E R T E D--->!
            !                    !  !  !  !  !  !  !  !
            !                    !  !  !  !  !  !  !  !
            v                    v  v  v  v  v  v  v  v
                      +--+--+--+--+--+--+--+--+--+--+--+--+
   Effective Address: |11|10| 9| 8| 7| 6| 5| 4| 3| 2| 1| 0|
                      +--+--+--+--+--+--+--+--+--+--+--+--+
                       ^  ^  ^                    ^  ^  ^
                        \  \  \                  /  /  /
         /^\             \  \  \                /  /  /
          !               \  \  \              /  /  /
          !                \  \  \            /  /  /
          !                 \  \  \          /  /  /
                             +--+--+--+--+--+--+--+--+
        Z Field              | 7| 6| 5| 4| 3| 2| 1| 0|
                             +--+--+--+--+--+--+--+--+
```
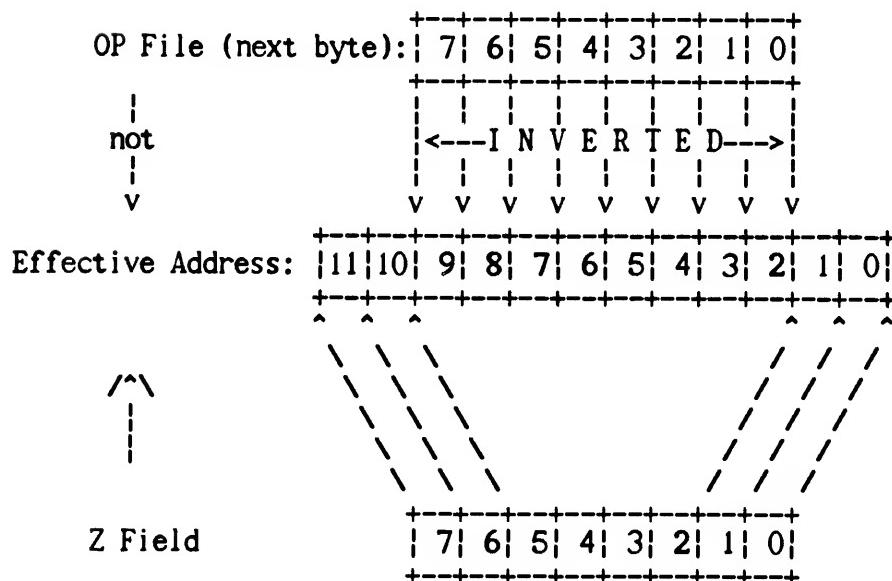
Dispatch   ·· 16 Way branch, based on the low-order four bits of the
                Shifter. Bits 5..2 come from the shifter (inverted);
                the remainder come from Z. This results in a 16 way
                branch with a spacing of four instructions.

```
                            +--+--+--+--+--+--+--+--+
        Shifter:        ...| 7| 6| 5| 4| 3| 2| 1| 0|
                            +--+--+--+--+--+--+--+--+
            !                           !  !  !  !
           not                          !-INVERTED- !
            !                           !  !  !  !
            v                           v  v  v  v  v
                      +--+--+--+--+--+--+--+--+--+--+--+--+
   Effective Address: |11|10| 9| 8| 7| 6| 5| 4| 3| 2| 1| 0|
                      +--+--+--+--+--+--+--+--+--+--+--+--+
                       ^  ^  ^  ^  ^  ^  ^              ^  ^  ^
                        \  \  \  \  \  \  \            /  /  /
         /^\             \  \  \  \  \  \  \          /  /  /
          !               \  \  \  \  \  \  \        /  /  /
          !                \  \  \  \  \  \  \      /  /  /
          !                 \  \  \  \  \  \  \    /  /  /
                             +--+--+--+--+--+--+--+--+
        Z Field              | 7| 6| 5| 4| 3| 2| 1| 0|
                             +--+--+--+--+--+--+--+--+
```

Vector   An up-to-eight way branch, based on the three
                micro-interrupt priority encoder bits (the V bits).
                Bits 4..2 come from the priority encoder (inverted),
                bit 5 is always zero, and the remaining bits come from
                Z. This results in an eight way branch with a spacing
                of four microinstructions.

```
                                                   +--+--+--+
Micro-Interrupt Priority Encoder:                  | 2| 1| 0|
                                                   +--+--+--+
      |                                            |  |  |  |
     not                                           |INVERTED|
      |                                      0      |  |  |  |
      v                                      v  v  v  v  v
           +--+--+--+--+--+--+--+--+--+--+--+--+
Effective Address: |11|10| 9| 8| 7| 6| 5| 4| 3| 2| 1| 0|
           +--+--+--+--+--+--+--+--+--+--+--+--+
            ^  ^  ^  ^  ^  ^  ^                 ^  ^  ^
             \  \  \  \  \  \  \           /  /  /
    /^\       \  \  \  \  \  \  \         /  /  /
     |         \  \  \  \  \  \  \       /  /  /
     |          \  \  \  \  \  \  \     /  /  /
                 \  \  \  \  \  \  \   /  /  /
                +--+--+--+--+--+--+--+--+
   Z Field      | 7| 6| 5| 4| 3| 2| 1| 0|
                +--+--+--+--+--+--+--+--+
```
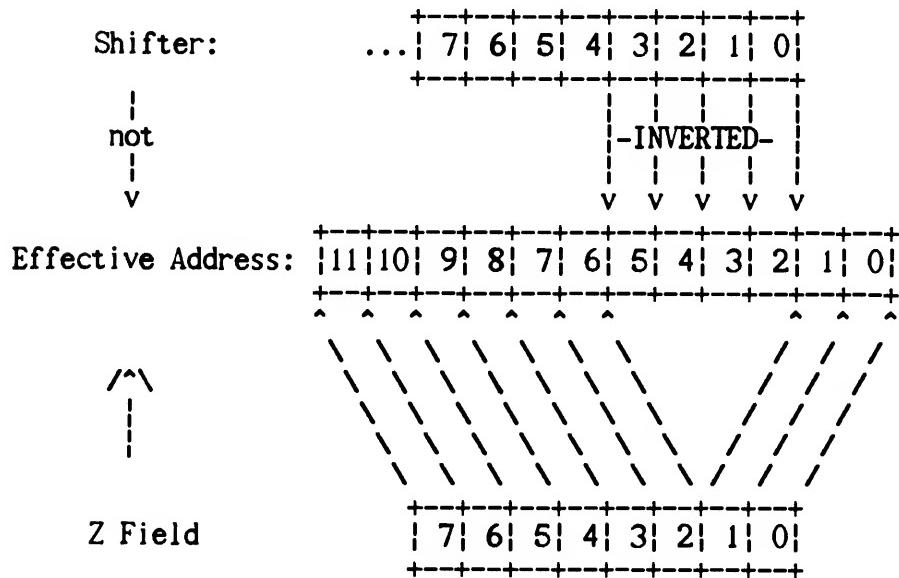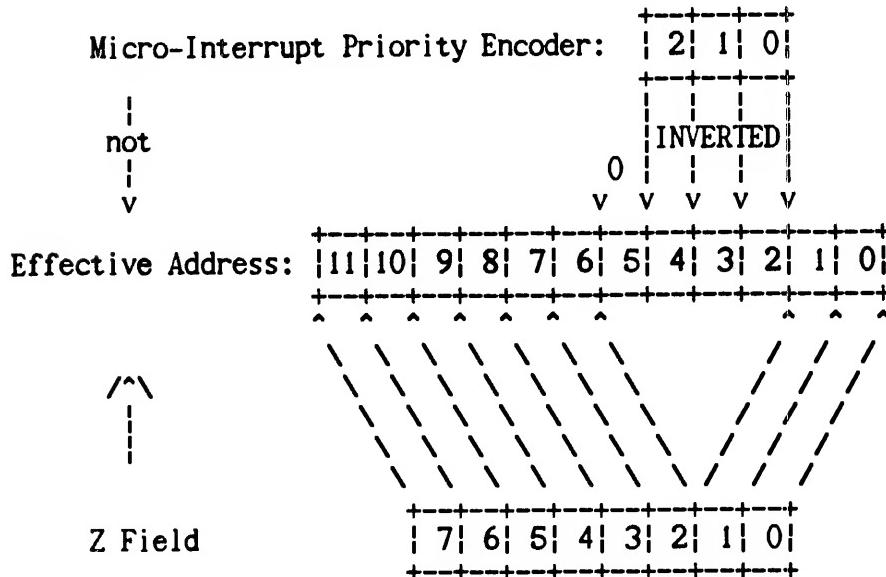
The syntax is:

    Jump = ['If' Condition] Directive ['(' Target ')'] .

        Condition = 'True' | 'False' | 'BPC[3]' | 'C19'
                  | 'IntrPend' | 'Odd' | 'ByteSign'
                  | 'Eql' | 'Neq' | 'Gtr' | 'Geq'
                  | 'Lss' | 'Leq' | 'Carry' | 'OverFlow'.

    Directive = 'Goto' | 'Call' | 'Return' | 'Next' | 'JumpZero'
              | 'LoadS' | 'GotoS' | 'CallS' | 'NextInst'
              | 'ReviveVictim' | 'PushLoad' | 'Vector' | 'Dispatch'
              | 'RepeatLoop' | 'Repeat' | 'JumpPop' | 'LeapPop'
              | 'Loop' | 'ThreeWayBranch' .

    Target = label | constant | 'Shift' .

3.4.1 Jump Conditions

Conditions are used to control whether or not a given jump is taken.
Some jump directives do not allow conditions, while for the remainder
the condition is optional. Section 3.4.2 presents in more detail
which directives may or may not allow condition execution.

All ALU related condition codes test the result of the ALU operation from the previous microcycle. Thus, the normal sequence is to perform an ALU operation and test its result in the next microinstruction. For example, comparison of two registers A and B could be done in this way:

```
A - B;
if gtr GOTO(Label);    !Jumps if A > B
```

All ALU tests with the exception of C19 test the lower 16 bits of the ALU. These are intended for data comparisons. After a subtraction, these condition codes compare the two operands. After other operations, these condition codes compare the 16-bit ALU result against zero.

The condition codes are not entirely sensible after a double precision add or subtract:

```
    ALower - BLower;
    AUpper - BUpper - OldCarry;
    if Lss Goto(AisLessThanB);
```

The Z condition code flag considers only the result of the upper precision operation and not the result of the lower precision operation. This means that EQL, NEQ, LEQ, and GTR condition codes (which use the Z flag) are not valid after a double precision add or subtract unless the low order 16 bits of the result are zero. Only LSS, GEQ, CARRY, and OVERFLOW reflect the result of the entire double precision operation. The following subroutine may be used to compare the two double precision numbers:

```
DblCmp: ALower - BLower;
        ALower - BLower - OldCarry,
            if Eql Return;    ! if ALower = BLower, the
                              ! condition codes are good
          not 0, if Lss Return;    ! A < B
          1, Return;               ! A > B
```

After calling DblCmp you may use the LSS, LEQ, EQL, NEQ, GEQ, and GTR condition codes. Carry and Overflow, however, are not valid.

C19 is designed for unsigned address comparisons. After an addition, it contains the INVERTED carry from bit 19; after a subtraction, it contains the borrow from bit 19.

Assuming that A and B are registers containing 20-bit addresses and T is a temporary register, the following code fragments show how C19 may be used to compare A and B.

```
A - 1;                  ! Jumps if A = 0;
if C19 GOTO(label);     ! doesn't jump if A <> 0

T := A;
T := T - B;
T - 1;                  ! Jumps if A = B;
if C19 GOTO(label);     ! doesn't jump if A <> B

A - B;                  ! Jumps if A < B;
if C19 GOTO(label);     ! doesn't jump if A >= B

B - A;                  ! Jumps if A > B;
if C19 GOTO(label);     ! doesn't jump if A <= B
```

The syntax is

```
condition  ::=   'If true'       |  'If false'
             |   'If BPC[3]'      |  'If C19'
             |   'If IntrPend'    |  'If Odd'
             |   'If ByteSign'    |  'If Eql'
             |   'If Neq'         |  'If Gtr'
             |   'If Geq'         |  'If Lss'
             |   'If Leq'         |  'If Carry'
             |   'If Overflow
```

3.4.1.1 True

Always Pass. Equivalent to no conditional.

3.4.1.2 False

Never Pass.

3.4.1.3 BPC[3]

Pass if BPC overflow has occurred (BPC[3] = 1). This implies that the Op file is empty.

3.4.1.4 C19

Pass if the inverted carry from ALU bit 19 of the last microinstruction was set. The sense of C19 is inverted, so if C19 is reset, a Pass occurs and if C19 is set, a Fail occurs.

3.4.1.5 IntrPend

Pass if any device is requesting an interrupt.

3.4.1.6 Odd

Pass if ALU bit 0 of the last microinstruction was set.

3.4.1.7 ByteSign

Pass if ALU bit 7 of the last microinstruction was set.

3.4.1.8 Eql

Pass if equal to zero. After a subtraction, this tests whether the two operands were equal (16-bit test). Otherwise, this tests whether the result was equal to zero (16-bit test).

3.4.1.9 Neq

Pass if not equal to zero. After a subtraction, this tests whether the two operands were unequal (16-bit test). Otherwise, this tests whether the result was unequal to zero (16-bit test).

3.4.1.10 Gtr

Pass if greater than zero. After a subtraction, this tests whether the A-input was greater than the B-input (16-bit test). Otherwise, this tests whether the result was greater than zero (16-bit test).

3.4.1.11 Geq

Pass if greater than or equal to zero, whether or not an overflow occurred. After a subtraction, this tests whether the A-input was greater than or equal to the B-input (16-bit test). Otherwise, this tests whether the result was greater than or equal to zero (16-bit test).

3.4.1.12 Lss

Pass if less than zero. After a subtraction, this tests whether the A-input was less than the B-input (16-bit test). Otherwise, this tests whether the result was less than zero (16-bit test).

3.4.1.13 Leq

Pass if less than or equal to zero. After a subtraction, this tests whether the A-input was less than or equal to the B-input 16-bit test). Otherwise, this tests whether the result was less than or equal to zero (16-bit test).

3.4.1.14 Carry

Pass if carry bit from bit 15 of the previous instruction was set. The ALU performs subtraction using two's complement arithmetic, and so the carry bit after a subtraction is a two's complement carry.

3.4.1.15 Overflow

Pass if a 16-bit overflow occurred in the previous microinstruction. Overflow is set under two conditions, as follows:

Sign of A, B Inputs	Operation	Result Sign
SAME	Addition	Different
DIFFERENT	Subtraction	Different from A

3.4.2 Jump Directives

Each directive specifies an action to take. Some directives do not allow a conditional to be specified (see section 3.4.1 for details on specifying the conditional). Some directives require a target to be specified, for some a target is optional, and for some the target is implied by the directive and cannot be explicitly specified. Targets are described in more detail in section 3.4.3.

The description for each directive includes whether or not a conditional and/or target can be supplied.

Further information about details of the jump instructions is available in the documentation for the AMD 2910 microsequencer.

The syntax is

```
Directive = 'Goto' | 'Call' | 'Return' | 'Next' | 'JumpZero'
          | 'LoadS' | 'GotoS' | 'CallS' | 'NextInst'
          | 'ReviveVictim' | 'PushLoad' | 'Vector' | 'Dispatch'
          | 'RepeatLoop' | 'Repeat' | 'JumpPop' | 'LeapPop'
          | 'Loop' | 'ThreeWayBranch' .
```

3.4.2.1 Goto

If the optional condition succeeds, or if no condition is supplied, branches to the specified target. If the optional condition fails, execution continues at the current instruction address plus one.

The condition is optional; the target is required.

3.4.2.2 Call

If the optional condition succeeds, or if no condition is supplied, pushes the current instruction address plus one onto the call stack and branches to the target. If the optional condition fails, execution continues at the current instruction address plus one.

The condition is optional; the target is required.

3.4.2.3 Return

If the optional condition succeeds, or if no condition is supplied, branches to the value of the top of the call stack and pops the call stack. If the optional condition fails, execution continues at the current instruction address plus one. The condition is optional; the target cannot be specified.

3.4.2.4 Next

Execution continues at the current instruction address plus one. The condition cannot be specified; the target cannot be specified.

3.4.2.5 JumpZero

Jumps to address zero in the control store.

The condition cannot be specified; the target cannot be specified.

3.4.2.6 Loads

Load the target into the S register. The S register is an address register internal to the microsequencer.

The condition cannot be specified; the target is required.

3.4.2.7 Gotos

There are three cases. If the optional condition succeeds (or no condition is supplied), and the optional target is supplied, branches to the target. If no target is supplied, branches to the next microinstruction in the source. If the condition fails, execution continues at the microinstruction at S.

The condition is optional if there is no target and required if there is a target. Unlike other jumps, the default condition is False which uses the value of the S register. The target is optional.


3.4.2.8 Calls

If the optional condition succeeds, or if no condition is supplied, branches to the target and pushes the current instruction address plus one onto the call stack. If the condition fails, pushes the current instruction address plus one onto the call stack and execution continues at control store address S.

The condition is optional if there is no target and required if there is a target. Unlike other jumps, the default condition is False which uses the value of the S register. The target is optional.

3.4.2.9 NextInst

A 256-way branch, based on the next byte from the Op file. The next byte provides bits 9..2 of the effective address (inverted); bits 11..10 come from Z[7:6] and bits bits 1..0 come from Z[1:0]. This results in a 256-way branch with a spacing of four instructions.

```
                                 +--+--+--+--+--+--+--+--+
         OP File (next byte): | 7| 6| 5| 4| 3| 2| 1| 0|
                                 +--+--+--+--+--+--+--+--+
                                 |  |  |  |  |  |  |  |  |
                                 |  |  |  |  |  |  |  |  |
            not                  |<---I N V E R T E D--->|
             |                   |  |  |  |  |  |  |  |  |
             |                   |  |  |  |  |  |  |  |  |
             v                   v  v  v  v  v  v  v  v  v
                              +--+--+--+--+--+--+--+--+--+--+--+--+
        Effective Address:  |11|10| 9| 8| 7| 6| 5| 4| 3| 2| 1| 0|
                              +--+--+--+--+--+--+--+--+--+--+--+--+
                              ^  ^  ^                    ^  ^  ^
                               \  \  \                  /  /  /
                                \  \  \                /  /  /
             /^\                 \  \  \              /  /  /
              |                   \  \  \            /  /  /
              |                    \  \  \          /  /  /
              |                     \  \  \        /  /  /
                                    +--+--+--+--+--+--+--+--+
         Z field                   | 7| 6| 5| 4| 3| 2| 1| 0|
                                    +--+--+--+--+--+--+--+--+
```
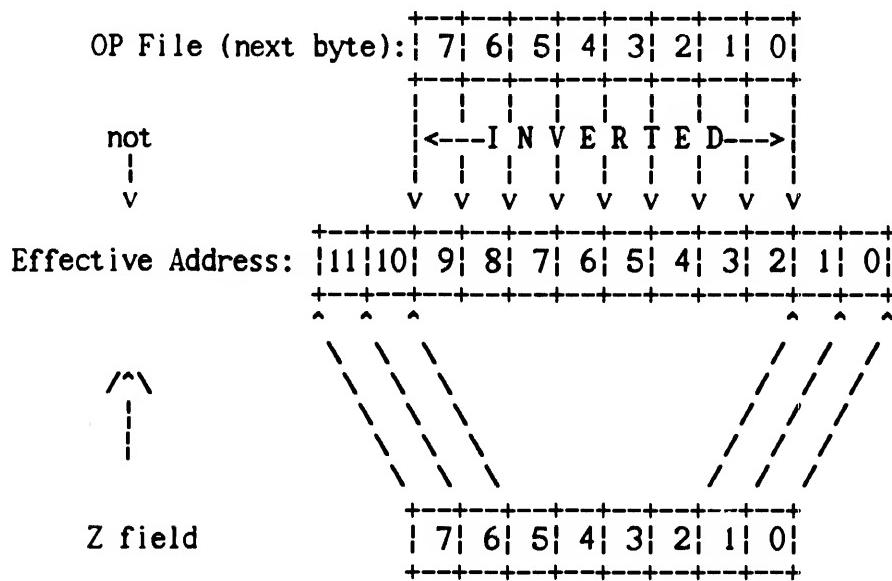
If the Op file is empty (BPC[3] = 1) then the OP File bits are forced to #377. Since these bits are inverted in the effective address, control passes to the base address generated from the Z bits (effective address bits 9..2 are zero). It is necessary to place a special refill function at this address ('Zaddr') which fetches four new bytes, increments UPC by four, performs a LOADOP (see section 3.5.1.4), and repeats the NEXTINST. The return from this special refill should be via NextInst.

The Z field is derived from the target by the assembler (see section 3.4.3).

The condition cannot be specified; the target is required.

3.4.2.10 ReviveVictim

Control branches to the value of VICTIM register. The victim register contains the address of the most recent microinstruction which included a NEXTOP while BPC[3] was set.

NOTE: The contents of the victim register are defined only from
        the time the nextOp is executed (with BPC[3] set) to the
        first execution of ReviveVictim. Therefore, REVIVEVICTIM
        can be executed only once after a failed NextOp.

The condition cannot be specified; the target cannot be specified.


3.4.2.11 PushLoad

If the optional condition succeeds or if no condition is supplied, execution continues at the current instruction address plus one, the current instruction address plus one is pushed onto the call stack, and the target is stored in S. If the condition fails, the current instruction address plus one is pushed onto the call stack and execution continues at that address.

The condition is optional; the target is required.

3.4.2.12 Vector

An up-to-eight way branch, based on the three micro-interrupt priority encoder bits (the V bits). Bits 4..2 come from the priority encoder (inverted), bit 5 is always zero, and the remaining bits come from Z. This results in an eight way branch with a spacing of four microinstructions.

```
                                              +--+--+--+
Micro-Interrupt Priority Encoder:  | 2| 1| 0|
                                              +--+--+--+
                                              |  |  |  |
    |                                         |  |  |  |
   not                                        |INVERTED|
    |                                     0   |  |  |  |
    v                                     v   v  v  v  v
              +--+--+--+--+--+--+--+--+--+--+--+--+
Effective Address:  |11|10| 9| 8| 7| 6| 5| 4| 3| 2| 1| 0|
              +--+--+--+--+--+--+--+--+--+--+--+--+
               ^  ^  ^  ^  ^  ^  ^              ^  ^  ^
                \  \  \  \  \  \  \        /  /  /
    /^\           \  \  \  \  \  \  \      /  /  /
     |             \  \  \  \  \  \  \    /  /  /
     |              \  \  \  \  \  \  \  /  /  /
     |               \  \  \  \  \  \  \/  /  /
                     +--+--+--+--+--+--+--+--+
  Z Field            | 7| 6| 5| 4| 3| 2| 1| 0|
                     +--+--+--+--+--+--+--+--+
```
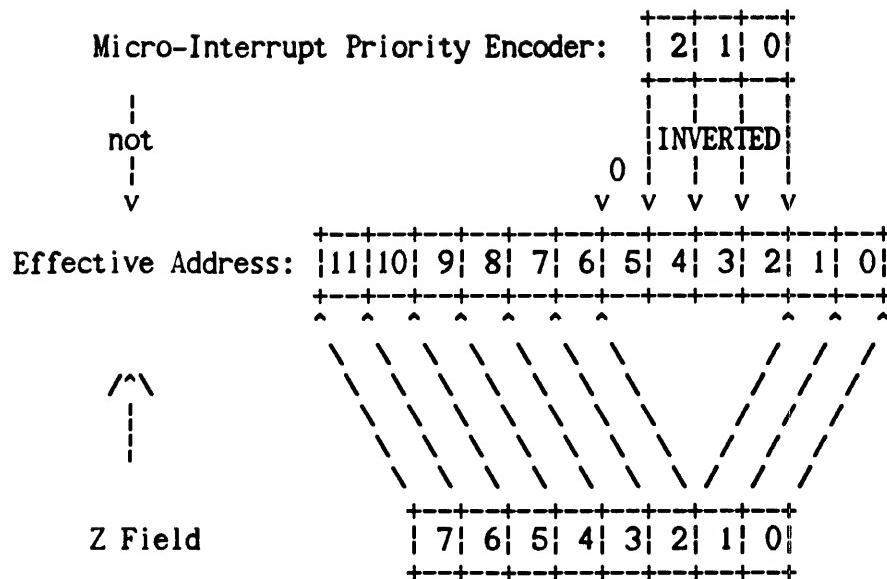
The microassembler derives the Z field from the target (see section 3.4.3).

If the optional condition succeeds, or if no condition is supplied, execution continues at the effective address. If the condition fails, execution continues at the current instruction address plus one. The condition is optional; the target is required.

3.4.2.13 Dispatch

A 16 Way branch, based on the low-order four bits of the Shifter. Bits 5..2 come from the shifter (inverted); the remainder come from Z. This results in a 16 way branch with a spacing of four instructions.

```
                               +--+--+--+--+--+--+--+--+
    Shifter:              ...| 7| 6| 5| 4| 3| 2| 1| 0|
                               +--+--+--+--+--+--+--+--+
          |                              |  |  |  |  |
          |                              |  |  |  |  |
        not                             |-INVERTED- |
          |                              |  |  |  |  |
          v                              v  v  v  v  v
                          +--+--+--+--+--+--+--+--+--+--+--+--+
    Effective Address: |11|10| 9| 8| 7| 6| 5| 4| 3| 2| 1| 0|
                          +--+--+--+--+--+--+--+--+--+--+--+--+
                           ^  ^  ^  ^  ^  ^  ^           ^  ^  ^
                            \  \  \  \  \  \  \       /  /  /
                             \  \  \  \  \  \  \     /  /  /
        /^\                   \  \  \  \  \  \  \   /  /  /
         |                     \  \  \  \  \  \  \ /  /  /
         |                      \  \  \  \  \  \ \/  /  /
                                 +--+--+--+--+--+--+--+--+
    Z Field                     | 7| 6| 5| 4| 3| 2| 1| 0|
                                 +--+--+--+--+--+--+--+--+
```
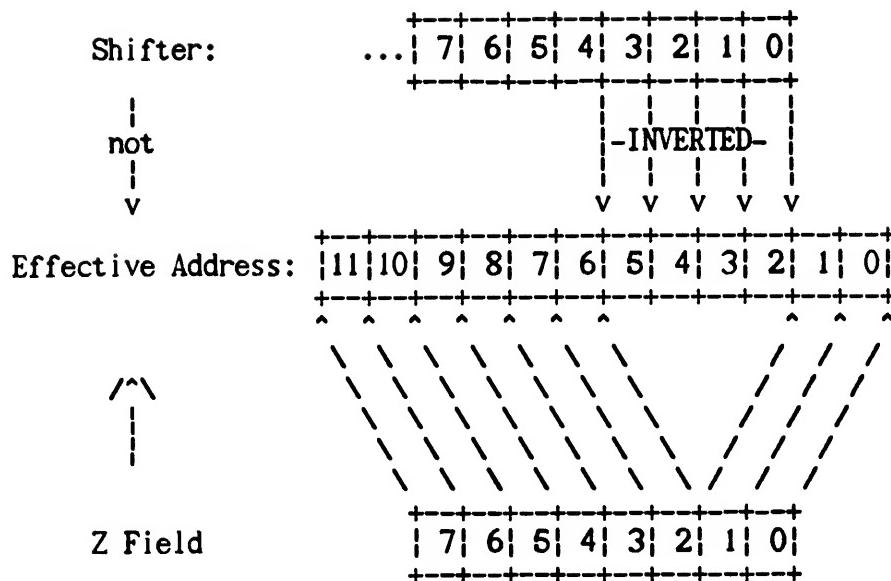
The Z field is derived from the target by the microassembler (see section 3.4.3 for details).

If the optional condition succeeds, or if no condition is supplied, execution continues at the effective address. If the condition fails, execution continues at the current instruction address plus one.

The condition is optional; the target is required.

3.4.2.14 RepeatLoop

If S is non-zero, execution continues at the instruction whose address is at the top of the call stack and S is decremented by one.

If S is zero, execution continues at the current instruction address plus one and the call stack is popped.

The conditional cannot be specified; the target cannot be specified.

3.4.2.15 REPEAT

If S is non-zero, execution continues at the target address and S is decremented by one.

If S is zero, execution continues at the current instruction address plus one.

The condition cannot be specified; the target is required.

3.4.2.16 JumpPop and LeapPop

If the optional condition succeeds, or if no condition is supplied, execution continues at the target address and the call stack is popped. If the condition fails, execution continues at the current instruction address plus one.

LeapPop is used on the PERQ1A CPU to pop both the upper and lower call stacks. LeapPop is encoded by the assembler as a JumpPop with the Hold bit set. See Section 3.4 for a description of the upper and lower call stacks of the PERQ1A.

The condition is optional; the target is required.

3.4.2.17 Loop

If the optional condition succeeds, or if no condition is supplied, execution continues at the current instruction address plus one. If the condition fails, then execution continues at the address specified by the top of the call stack and the call stack is popped.

The condition is optional; the target cannot be specified.

3.4.2.18 ThreeWayBranch

If the optional condition succeeds, or no condition is supplied, then execution continues at the current instruction address plus one. The call stack is popped, and if S is non-zero, S is decremented.

If the condition is supplied and fails, then if S is non-zero, execution continues at the address specified by the top of the call stack and S is decremented. If S is zero, execution continues at the target address and the call stack is popped.

The condition is optional; the target is required.

3.4.3 Targets

Targets are used by the microassembler to derive the appropriate contents for the F, SF, Z fields. The microassembler determines whether or not a jump is long or short and sets the F and SF fields accordingly.

The syntax is

target   =   label | constant | 'Shift' .

3.4.3.1 Label

A label is formed from a letter followed by an arbitrary number of letters or digits. The microassembler will use, as the target address of the jump, the control store address of the instruction labelled with the corresponding name (see section 2.1 for details of microinstruction labelling).

3.4.3.2 Constant

A constant is either an explicit constant contructed from a sequence of digits (optionally preceded by a radix indicator) or it may be a previously defined name (see section 3.1.2).

3.4.3.3 Goto(Shift)

On the PERQ1A, the shifter outputs may be used as the address by typing the word "Shift" where a jump address is allowed. The shifter may be used with the following jump types: Call, Goto, PushLoad, CallS, GotoS, Repeat, JumpPop, LeapPop, and LoadS. The following example jumps to an address in a register named Addr.

```
Addr, RightShift(0);            ! run address through shifter
Goto(Shift);                    ! jump
```

The following example performs an n-way dispatch (n <= 4096) to address Addr + N * 4.

```
N, LeftShift(2);                ! multiply N by 4 by shifting
Shift + Addr, RightShift(0);    ! add base and send to shifter
Goto(Shift);                    ! jump
```

3.5 SPECIAL FUNCTIONS

Memory references, Writable Control Store (WCS) references, certain rasterOp control functions, and several housekeeping operators are handled by the microassembler as special functions.

Special functions requiring no arguments (Nonary functions) are described in section 3.5.1, special functions requiring one argument are described in section 3.5.2, and special functions requiring two arguments are described in section 3.5.3.

The syntax is

    Special = Nonary | Unary | Binary .

      Nonary = 'WCSlow' | 'WCSmid' | 'WCShi' | 'LoadOp' | 'Hold'
            | 'StackReset' | 'Push' | 'Pop' | 'Fetch' | 'Fetch2'
            | 'Fetch4' | 'Fetch4R' | 'Store' | 'Store2'
            | 'Store4' | 'Store4R' | 'LatchMA' | 'ShiftOnR' .
            | 'MultiplyStep' | 'DivideStep' .

    Unary = UnaryName '(' ConstExpr ')' .

      UnaryName = 'LeftShift' | 'RightShift' | 'Rotate' | 'IOB'
          | 'CntlRasterOp' .

    Binary = BinaryName '(' ConstExpr ',' ConstExpr ')' .

    BinaryName = 'Field' .

3.5.1 Nonary

Nonary functions require no arguments. It should be noted that the memory reference functions (the fetch-type and store-type functions) require that specific timing constraints be satisfied. Other constraints or specific timing requirements are detailed in the relevant section.

The memory system cycles in 680 nanoseconds (exactly four microcycles). Microcycles are numbered starting at 0, and denoted 't0', 't1', 't2', and 't3'. Requests must be made in a particular cycle (depending on the type of the request). If a memory request is made in the wrong cycle, the processor will be suspended until the correct cycle, or, in some cases, the improper request will be ignored altogether. In the discussions which follow, 'fetch' or 'store' refer to a memory function which fetches or stores exactly one word. The generic terms 'fetch type' and 'store type' refer to any fetching or storing reference.

There are eight types of memory references, coded into the SF field while F = 1. They are encoded as follows:

SF	Type	Description
10	Fetch4R	Fetch four words, transport in reverse order
11	Store4R	Store four words, transport in reverse order
12	Fetch4	Fetch four words
13	Store4	Store four words
14	Fetch2	Fetch two words
15	Store2	Store two words
16	Fetch	Fetch one word from memory
17	Store	Store one word into memory

The address for all memory references comes from R. For all fetch type references, the address (and the request itself) is latched at t3 and data is available from MDI or MDX at the following t2. If MDI or MDX is used during a t0 or t1 immediately following a fetch type memory reference, the processor is suspended until t2.

Any address may be used with a Fetch, and the memory word may be read during any cycle from t2 until the following t1.

The low-order bit of the address for a Fetch2 is ignored, so that a Fetch2 is always double-word aligned. After a Fetch2, the first word must be read at t2, and the second word must be read at t3.

The two low-order bits of the address for a Fetch4 or Fetch4R are ignored, so that a Fetch4 or Fetch4R is always quad-word aligned. After a Fetch4 or Fetch4R, the first word must be read at t2, the second at t3, the third at the next t0, and the last at the next t1. Fetch4R returns word3 of the quad-word first, then word2, word1, and word0. This word reversal (from the fetch4 sequence) is primarily useful for RasterOp so that it can do left to right as well as right to left transfers.

Any address may be used with a Store. The address and Store command are given in a t2 cycle and the data to be written is supplied on R in the following t3.

The low-order bit of the address for a Store2 is ignored, so that a Store2 is always double-word aligned. The address and Store2 are given in a t3 cycle, and the data is supplied on R during the following t0 and t1.

The two low-order bits of the address for a Store4 or Store4R are ignored, so that a Store4 or Store4R is always quad-word aligned. The address and Store4 are given in a t3 cycle, and the data is supplied in the four following cycles (t0, t1, t2, t3). Store4R stores word3 of the quad-word first, then word2, word1, and word0. This word reversal (from the Store4 sequence) is primarily useful for RasterOp so that it can do left to right as well as right to left transfers.

The following are examples of each type of reference and their code:

```
Fetch:          MA := Addr, Fetch;        (t3)
                ...                       (t0)
                ...                       (t1)
                Data := MDI;              (t2)

Fetch2:         MA := Addr, Fetch2;       (t3)
                ...                       (t0)
                ...                       (t1)
                Data0 := MDI;             (t2)
                Data1 := MDI;             (t3)

Fetch4:         MA := Addr, Fetch4;       (t3)
                ...                       (t0)
                ...                       (t1)
                Data0 := MDI;             (t2)
                Data1 := MDI;             (t3)
                Data2 := MDI;             (t0)
                Data3 := MDI;             (t1)

Fetch4R:        MA := Addr, Fetch4R;      (t3)
                ...                       (t0)
                ...                       (t1)
                Data3 := MDI;             (t2)
                Data2 := MDI;             (t3)
                Data1 := MDI;             (t0)
                Data0 := MDI;             (t1)

Store:          MA := Addr, Store;        (t2)
                MDO := Data;              (t3)

Store2:         MA := Addr, Store2;       (t3)
                MDO := Data0;             (t0)
                MDO := Data1;             (t1)

Store4:         MA := Addr, Store4;       (t3)
                MDO := Data0;             (t0)
                MDO := Data1;             (t1)
                MDO := Data2;             (t2)
                MDO := Data3;             (t3)

Store4R:        MA := Addr, Store4R;      (t3)
                MDO := Data3;             (t0)
                MDO := Data2;             (t1)
                MDO := Data1;             (t2)
                MDO := Data0;             (t3)
```

The IO system can request memory cycles at any time. The memory system gives priority to the IO system so that if both the processor and the IO system make memory requests, the IO is served first while the processor is delayed. The Hold bit, if set, locks out IO requests

while it is set. To be effective, Hold must be asserted in a t2. This is necessary only when doing overlapped memory references.

In some contexts, a request made in an improper cycle will be ignored as follows:

1. After a Fetch or Fetch2 (in t3), any memory reference in t0 or t1 is ignored. A Store specified in the t2 will start immediately, but all others will abort until the correct time.

2. Fetch4 and Fetch4R follow the rules for Fetch and Fetch2 with the exception that a Store4 (in the same direction--forward or reverse) can be specified in t0, but this is only used for RasterOp.

3. After a Store (in t2), any memory reference in t3 or t0 is ignored. References started in t1 are aborted until the correct cycle.

4. After a Store2, Store4 or Store4R (in t3), any memory reference in t0 through t3 is ignored. Memory references started in t0 are aborted until the correct cycle.

5. To be effective, Hold must be asserted in a t2. You must be careful about aborts caused by using MDI in the wrong cycle--you may be aborted past the t2, causing the Hold to be ignored. You may not specify Hold too often--you must allow an IO reference at least once in every 3 memory cycles.

6. After a Fetch, MDI is valid from t2 through the following t1 (four full cycles). For Fetch2, Fetch4, and Fetch4R, each MDI is valid for a single microcycle.

These six constraints can be simplified into the following two simple rules. These two rules, if followed, will never cause a problem, but they may preclude certain performance optimizations permitted under rigorous application of the above six constraints. The simple guidelines are:

A. Never start a memory reference after a fetch type reference until you have taken all the data.

B. Never start a memory reference during the four microinstructions which follow a store type request.

Following these rules, we can construct many interesting overlapped memory requests. Note that in the following examples, Hold is always asserted in a t2. A Fetch ... Store sequence is an exception--you need not use Hold, but it doesn't hurt performance, so we assert it for consistency.

Indirect fetches:

```
MA := Addr, Fetch;              (t3)
instruction or Nop;            (t0)  must be explicit
instruction or Nop;            (t1)  must be explicit
MA := MDI, Fetch<n>, Hold;     (t2, t3)  any type of fetch
    ...
Data := MDI;                    (t2)
    ...
```

Hold is asserted in t2 so that IO requests do not pre-empt
the processor. The instruction "MA := MDI, Fetch<n>, Hold;"
first tries to execute in t2, but is aborted until t3
because it contains a fetch. The MDI is still valid because
MDI is valid from t2 to the following t1 after a Fetch.

```
MA := Addr, Fetch;              (t3)
instruction or Nop;            (t0)  must be explicit
instruction or Nop;            (t1)  must be explicit
instruction, Hold;             (t2)
MA := MDI, Fetch<n>;           (t3)  any type of fetch
    ...
Data := MDI;                    (t2)
    ...
```

Again, Hold is asserted in t2. Note that this differs from
the previous example in that the Hold and Fetch<n> are not
done in the same instruction. These two examples show that
for indirect fetches, the two fetches may be separated by
two or three other instructions.

Indirect stores:

```
MA := Addr, Fetch;              (t3)
instruction or Nop;            (t0)  must be explicit
instruction or Nop;            (t1)  must be explicit
MA := MDI, Store, Hold;        (t2)
MDO := Data;                    (t3)
```

In this case, the MDI, the Store, and the Hold all execute
in t2.

```
MA := Addr, Fetch2;             (t3)
instruction or Nop;            (t0)  must be explicit
instruction or Nop;            (t1)  must be explicit
MA := MDI, Store, Hold;        (t2)
MDO := MDI;                     (t3)
```

In this case, the first fetched word is used as an address,
and the second is used as data to be stored.

```
MA := Addr, Fetch;          (t3)
instruction or Nop;         (t0)  must be explicit
instruction or Nop;         (t1)  must be explicit
MA := MDI, Store<n>, Hold;  (t2, t3)  any except Store
MDO := Data;                (t0)
   ...
```

Hold is asserted in t2 so that IO requests do not pre-empt the processor. The instruction "MA := MDI, Store<n>, Hold;" first tries to execute in t2, but is aborted until t3 because it contains a store. The MDI is still valid because MDI is valid from t2 to the following t1 after a Fetch.

```
MA := Addr, Fetch;          (t3)
instruction or Nop;         (t0)  must be explicit
instruction or Nop;         (t1)  must be explicit
instruction, Hold;          (t2)
MA := MDI, Store<n>;        (t3)  any except Store
MDO := Data;                (t0)
   ...
```

Again, Hold is asserted in t2. Note that this differs from the previous example in that the Hold and Store<n> are not done in the same instruction. These two examples show that for indirect stores, the Fetch and the Store<n> may be separated by two or three other instructions.

Copy operations:

```
MA := Addr1, Fetch;         (t3)
instruction or Nop;         (t0)  must be explicit
instruction or Nop;         (t1)  must be explicit
MA := Addr2, Store, Hold;   (t2)
MDO := MDI;                 (t3)
```

A word is copied from one memory location to another. Unfortunately, two or four word copies are not possible because the times when data must be read and written are different for the fetches and stores.

3.5.1.1 WCS Directives

The WCS directives are used to write locations within the writeable control store. The microinstruction word is 48 bits long, so each WCS word is written as three 16-bit words, using WCSLow, WCSMid and WCSHi. In order to write a location of the controlstore, the address which is to be written should be placed in the S register with a LoadS instruction. The WCSLow, WCSMid, and WCSHigh functions are executed

in three successive microinstructions together with the jump code
GotoS. During each one of these WCS functions, 16 bits of a
microinstruction is written with the value of the ALU result from the
previous micro- instruction. It is important that the jump codes be
written correctly, as it is this part of the microinstruction that
supplies the address.

Each instruction which contains a WCSLow, WCSMid, or WCSHigh takes 340
nanoseconds to complete because it is executed twice. This precludes
instructions which change their own initial conditions such as

       R := R + 1, WCSLow

The following example writes a microinstruction with the values in
three registers. The example does not show values being loaded into
the registers, but this must be done prior to the execution of the
WCSLow, WCSMid, and WCSHi functions.

       Define(LowWord, 100);
       Define(MidWord, 101);
       Define(HighWord, 102);

       LowWord, LoadS(MicroAddress);
       MidWord, WCSLow, if True GotoS(A); A: HighWord, WCSMid, if True
GotoS(B); B: WCSHi, if True GotoS(C); C:

Note that since each directive takes two microcycles to complete, a
total of at least 6 microcycles is needed to write any location in the
writeable control store.

The WCSLow directive causes the contents of R to be written into the
low order 16 bits of a location in the Writeable Control Store
(WCS[15:0]).

Note: The WCSlow directive requires two microcycles to complete.

The WCSmid directive causes the contents of R to be written into the
middle 16 bits of a location in the Writeable Control Store
(WCS[31:16]).

Note: The WCSmid directive requires two microcycles to complete.

The WCShi directive causes the contents of R to be written into the
high order 16 bits of a location in the Writeable Control Store
(WCS[47:32]).

Note: The WCShi directive requires two microcycles to complete.

3.5.1.2 LoadOp

The PERQ provides special hardware to speed up the filling of the Op

file (in order to improve the performance of refill). After a Fetch4, a LoadOp specified in the t1 immediately following the fetch4 causes the hardware to load four words into the Op file without further microcode assistance.

The LoadOp should be given in the t1 immediately following the Fetch4. The instruction which follows the LoadOp can go back to the NextInst/NextOp since the first byte is guaranteed to be in. The three remaining words arrive and are placed in the Op file without microcode assistance.

Note: If BPC is non-zero (to start reading in the middle of the quadWord), the refill code must wait until the correct byte is in the Op file.

3.5.1.3 Hold

The HOLD function is used to inhibit IO devices from accessing memory. The IO system can request memory cycles at any time. The memory system gives priority to the IO system so that if both the processor and the IO system make memory requests, the IO is served first while the processor is delayed. IO requests are locked out during the execution of a microinstruction with the hold function set. To be effective, hold must be set in a T2. This is necessary only while doing overlapped memory references (see sections 3.5.1.9 through 3.5.1.15).

The hold function must not be set all the time; an IO reference must be permitted at least once every three memory cycles.

3.5.1.4 StackReset

The StackReset special function causes the expression stack to be emptied.   The StackEmpty (SE) bit in the microstatus word (USTATE[8]) is asserted (to the zero state) when this function is executed, and the DDS is incremented.

Note: The result of operations other than 'push' which reference the stack while the stack is empty are implementation-dependent and should be considered undefined.

3.5.1.5 Push

The Push special function causes the expression stack pointer to be incremented.   Data is placed on the stack through an assignment using the Push special function, as follows:

TOS := Data, Push;   ! Pushes 'data' on the stack.

A Push into the StackTop (the 16'th push) causes the stack empty bit to become set (to the zero state). This indicates that subsequent pushes (without intervening pops) produce undefined results.

Note: The result of a push while the stack is full is implementation-dependent and should be considered undefined.

3.5.1.6 Pop

The Pop special function causes the expression stack poiner to be decremented. One or more pops while the stack is empty (with no intervening pushes) produces undefined results.

Note: The result of pop while the stack is empty is implementation dependent and should be considered undefined.

3.5.1.7 Fetch

The Fetch special function initiates a one word memory data read sequence. The memory address for a Fetch is latched from the contents of the R register at the time that the Fetch is recognized (t3).

The data from a fetch is available from MDI or MDX at the next t2. If MDI or MDX is used during the intervening t0 and t1, the processor is suspended until t2.

Any address may be used with a Fetch, and the resulting data word may be read in any cycle from t2 until the following t1 (inclusive).

3.5.1.8 Fetch2

The Fetch2 special function initiates a double word memory data read sequence. The memory address for a Fetch2 is latched from the contents of the R register at the time that the Fetch2 is recognized (t3).

The first data word returned from a Fetch2 (word 0 of the double word) must be read at the next t2; the second word (word 1 of the double word) must be read at the next t3. The state of MDI and MDX is undefined in succeeding cycles. If MDI or MDX is used during the intervening t0 and t1 (after a Fetch2 at t3) the processor is suspended until t2.

The low-order address bit of a Fetch2 request is ignored, so that a Fetch2 is always double-word aligned (rounded down).

3.5.1.9 Fetch4

The Fetch4 special function initiates a quad-word memory data read sequence. The memory address for a Fetch4 is latched from the contents of the R register at the time that the Fetch4 is recognized (t3).

The first data word returned from a Fetch4 (word 0 of the quad word) must be read at the next t2; the second word (word 1 of the quad word) must be read at the next t3, the third (word 2 of the quad word) at t0, and the last (word 4 of the quad word) at t1. The state of MDI and MDX is undefined in succeeding cycles.

The two low-order address bits of a Fetch4 request are ignored, so that a Fetch4 is always quad-word aligned (rounded down).

3.5.1.10 Fetch4R

The Fetch4R special function initiates a quad-word memory data read sequence. The Fetch4R is exactly like the Fetch4, with the exception that the quad word is returned in reverse order. Thus, the high-order word is received first and the low-order word last. The memory address for a Fetch4R is latched from the contents of the R register at the time that the Fetch4R is recognized (t3).

The first data word returned from a Fetch4R (word 3 of the quad word) must be read at the next t2; the second word (word 2 of the quad word) must be read at the next t3, the third (word 1 of the quad word) at t0, and the last (word 0 of the quad word) at t1. The state of MDI and MDX is undefined in succeeding cycles.

The two low-order address bits of a Fetch4R request are ignored, so that a Fetch4 is always quad-word aligned (rounded down).

3.5.1.11 Store

The Store special function initiates a one word memory data write sequence. The memory address for a Store is latched from the contents of the R register at the time that the Store is recognized (t2).

Any address may be used with a Store. The data for a Store is supplied on R in the following t3.

3.5.1.12 Store2

The Store2 special function initiates a double word memory data write sequence. The memory address for a Store2 is latched from the contents of the R register at the time that the Store2 is recognized (t3).

The low-order bit of the address for a Store2 is ignored, so a Store2
always writes to a double-word aligned location (rounded down). The
first data word for a Store2 (word 0 of the double word) is supplied
on R during the next t0 and the second data word (word 1 of the double
word) is supplied on R during the following t1.

3.5.1.13 Store4

The Store4 special function initiates a quad-word memory data write
sequence.   The memory address for a Store4 is latched from the
contents of the R register at the time that the Store4 is recognized
(t3).

The two low-order bits of the address for a Store4 are ignored, so a
Store4 always writes to a quad-word aligned location (rounded down).
The first data word for a Store4 (word 0 of the quad-word) is supplied
on R during the next t0, the second data word (word 1 of the
quad-word) is supplied on R during the following t1, the third data
word (word 2 of the quad-word) is supplied on R during the following
t2, and the last data word (word 3 of the quad-word) is supplied on R
during the following t3.

3.5.1.14 Store4R

The Store4R special function initiates a quad-word memory data write
sequence.   The Store4R is exactly like the Store4, with the exception
that the quad word is written in reverse order. Thus, the high-order
word is written first and the low-order word last. The memory address
for a Store4R is latched from the contents of the R register at the
time that the Store4R is recognized (t3).

The two low-order bits of the address for a Store4R are ignored, so a
Store4R always writes to a quad-word aligned location (rounded down).
The first data word for a Store4R (word 3 of the quad-word) is
supplied on R during the next t0, the second data word (word 2 of the
quad-word) is supplied on R during the following t1, the third data
word (word 1 of the quad-word) is supplied on R during the following
t2, and the last data word (word 0 of the quad-word) is supplied on R
during the following t3.

3.5.1.15 ShiftOnR

The ShiftOnR special function, by obtaining the shift control from the
R register, allows a shift function to be a variable.

The shifter hardware can either rotate a 16-bit item 0 to 15 places
(rightward), can right justify an arbitrarily positioned 0 to 15 bit
subfield of an item, or it can right or left shift a 16-bit item 0 to

15 places.

The shifter hardware is driven by two four bit nibbles, encoded as follows (R has a 16 bit item to be shifted):

a) Mask Operations:

    m := Mask Width (bits, one origin)
          $1 <= m <= 16$
    i := Index of low-order bit of Mask
          $0 <= i <= 15$

    For i <= 16 - m,

        Result = {R and [$(2^m - 1) * 2^i$]}

            Low-order Nibble := m-1
            High-order Nibble := i

b) LeftShift Operations:

    j := Number of places to shift, $0 <= j <= 15$

    Low-order Nibble := #17
    High-order Nibble := j

c) Rotate Operations:

    j := Number of places to rotate, $0 <= j <= 15$

    For $0 <= j <= 7$,

        Low-order Nibble := #15
        High-order Nibble := j

    For $8 <= j <= 15$,

        Low-order Nibble := #16
        High-order Nibble := j + 8

d) RightShift Operations

    n := Number of places to RightShift,
          $0 <= n <= 15$

        Low-order Nibble := 15 - n
        High-order Nibble := n

The usage sequence for the ShiftOnR special function is:

  1. Put the shift control byte on R and execute ShiftOnR.

3 - 43

2. Put the item to be shifted on R.

3. Read the shifted result on shift. The shifter control
    logic always keeps the last shift control function loaded
    so that the shifter can shift a succession of words without
    respecifying the shift control function. The shift ouputs
    always have the shifted value of what was last on R.

3.5.1.16 MultiplyStep

The PERQ1A has hardware to support single precision multiplication and
division. The multiply/divide hardware consists of a 16-bit shift
register called MQ and a control circuit. The function of the
multiply/divide hardware (off, signed multiply, unsigned multiply,
unsigned divide) is controlled by 2 bits in the WidRasterOp register:

WidRasterOp<7:6>	Operation
0	Off
1	Unsigned Divide
2	Unsigned Multiply
3	Signed Multiply

Signed divides are faked by remembering the signs of the dividend and
divisor. The divide is performed on the absolute values of the
dividend and divisor. The resulting quotient and remainder are
negated if necessary.

At each step of a multiplication, the multiplicand is added to the
partial product if the least significant bit of the multiplier is set.
The partial product and the multiplier are then shifted to the right.
Sixteen steps are needed to compute the full product.

✱ The MQ register is initially loaded with the multiplier by the "MQ
  :=" special function. This may be done before or after the multiply
  hardware is enabled.

✱ The ALU add and subtract functions (without OldCarry) perform
  normally if the low order bit of MQ is set, but if it is not set,
  the ALU passes the AMux value through unchanged. Thus, the adding
  or subtracting of the multiplicand is done only if the corresponding
  bit of the multiplier is set.

✱ A Rotate(1) shift function should be used. When Shift is used as
  the AMux source, Shift<15> contains Result<15> Xor Overflow from the
  previous instruction for signed multiply or contains the Carry from
  the previous instruction for unsigned multiply. The remaining bits
  of Shift have their normal values.

� When the MultiplyStep special function is executed, the MQ register
is shifted to the right and Result<0> from the current instruction
is shifted into MQ<15>. Thus, the low precision word of the product
is shifted into MQ.

✱ The upper precision word of the product is left in an XY register.

✱ The low precision word of the product is read by using the ":= MQ"
special function which forces MQ onto R. This value does not pass
through the ALU, thus no computation is possible during this cycle.
Reasonable actions are 1) to write the value of MQ into a register
and 2) push MQ onto the EStk and send MQ to the processor shifter.
Note that since the value does not pass through the ALU, condition
codes are invalid in the cycle which follows reading MQ.

The following example performs signed multiplication of two single
precision numbers yielding a double precision product. Note that the
last MultiplyStep is done with a subtract rather than an add. This is
because we interpret the multiplier as:

MQ<0> ✱ 2^0 + MQ<1> ✱ 2^1 + ... + MQ<14> ✱ 2^{14} - MQ<15> ✱ 2^{15}

```
Constant(OffMultiply, 0);
Constant(OffDivide, 0);
Constant(UnSignedDivide, 100);
Constant(UnSignedMultiply, 200);
Constant(SignedMultiply, 300);

Define(Multiplier, 200);
Define(Multiplicand, 201);
Define(ProductLow, 202);
Define(ProductHigh, 203);

Rotate(1);                              ! shifter must rotate right 1
MQ := Multiplier;                       ! load the multiplier
WidRasterOp := SignedMultiply;          ! set signed multiply
ProductHigh := 0,                       ! partial product to shifter
            PushLoad(10#14);            ! push .+1, set S to 10#14
Shift + Multiplicand, MultiplyStep, RepeatLoop; ! 10#15 steps
Shift - Multiplicand, MultiplyStep;     ! 10#16th step for sign bit
ProductHigh := Shift;                   ! read upper precision product
ProductLow := MQ;                       ! read lower precision product
WidRasterOp := OffMultiply;             ! turn off multiply hardware
```

During unsigned multiply all multiply steps use addition. This is because we interpret the multiplier as:

$$MQ<0> * 2^0 + MQ<1> * 2^1 + \ldots + MQ<14> * 2^{14} + MQ<15> * 2^{15}$$

```
ProductHigh := 0,                      ! partial product to shifter
                PushLoad(10#15);! push .+1, set S to 10#15
Shift + Multiplicand, MultiplyStep, RepeatLoop; ! 10#16 steps
ProductHigh := Shift;                  ! read upper precision product
ProductLow := MQ;                      ! read lower precision product
```

3.5.1.17 DivideStep

The multiply/divide hardware can also be used to do unsigned division by a non-restoring division algorithm. At each step, the divisor is subtracted (or added) from the partial remainder and a new bit of the dividend is shifted left into the partial remainder. Rather than restoring the partial remainder after subtracting (or adding) too much, the divisor is added (or subtracted) on the next step.

✻ The MQ register is initially loaded with the dividend by the "MQ :=" special function. This may be done before or after the divide hardware is enabled.

✻ The ALU subtract function (without OldCarry) performs normally if Result<15> of the previous instruction was not set, but if it was set, an add (without OldCarry) is performed instead. Thus the subtracting or adding of the divisor is controlled by the sign bit of the previous ALU result.

✻ A Rotate(10#15) shift function should be used. When Shift is used as the AMux source, Shift<0> contains the value that MQ<15> had at the beginning of the previous instruction. Thus the dividend is shifted into the AMux from the right.

✻ When the DivideStep special function is executed, the MQ register is shifted to the left and the complement of Result<15> from the current instruction is shifted into MQ<0>. Thus the quotient is shifted into MQ.

✻ The quotient is read 16 bits at a time by the ":= MQ" special function which forces the lower precision quotient onto R. This value does not pass through the ALU, thus no computation is possible during this cycle. Reasonable actions are 1) to write the value of MQ into a register and 2) push MQ onto the EStk and send MQ to the processor shifter. Note that since the value does not pass through the ALU, condition codes are invalid in the cycle which follows reading MQ.

✻ The remainder is left in an XY register. Since a non-restoring algorithm is used, the loop may terminate with a negative remainder.

In this case, the divisor is added back into the remainder.

The following example performs a full single precision divide of a double precision dividend and a single precision divisor yielding a double precision quotient and a single precision remainder. The division proceeds in two parts each of which does 16 bits of the division.

```
        Constant(OffMultiply, 0);
        Constant(OffDivide, 0);
        Constant(UnSignedDivide, 100);
        Constant(UnSignedMultiply, 200);
        Constant(SignedMultiply, 300);

        Define(DividendLow, 200);
        Define(DividendHigh, 201);
        Define(Divisor, 202);
        Define(QuotientLow, 203);
        Define(QuotientHigh, 204);
        Define(QuotientSign, 205);
        Define(RemainderSign, 206);

        Tos := 0, Push;                        ! 0 for two's complementing
        RemainderSign := DividendHigh, RightShift(0); !set remainder sign
        QuotientSign := Shift xor Divisor,     ! set sign of quotient
                                        if Geq Goto(A);
        DividendLow := Tos - DividendLow;! abs value of dividend
        DividendHigh := Tos - DividendHigh - OldCarry;
        Divisor;
A:  if Geq Goto(B);                     ! if divisor >= 0
        Divisor := Tos - Divisor;          ! abs value of divisor
B:  Rotate(10#15);                      ! shifter must rotate left 1
        MQ := DividendHigh;                ! load upper dividend
        WidRasterOp := UnSignedDivide;     ! set unsigned divide
        LoadS(10#15);                      ! S := 10#15
        Remainder := 0,                    ! initialize partial remainder
                        DivideStep;        ! get started
C:  Remainder := Shift - Divisor, DivideStep, Repeat(C); ! 10#16 steps
        QuotientHigh := MQ;                ! read upper quotient
        MQ := DividendLow;                 ! load lower dividend
        LoadS(10#15);                      ! S := 10#15
        Remainder,                  ! send remainder conditional subtract
                        DivideStep;        ! get started
D:  Remainder := Shift - Divisor, DivideStep, Repeat(D);! 10#16 steps
        WidRasterOp := OffDivide,          ! turn off divide hardware
                        if Geq Goto(E); ! if remainder >= 0
        Remainder := Remainder + Divisor;! correct remainder
```

```
E: QuotientLow := MQ;                    ! read lower quotient
     QuotientSign;
     RemainderSign, if Geq Goto(F);      ! if quotient should be >= 0
     QuotientLow := Tos - QuotientLow;   ! set negative quotient
     QuotientHigh := Tos - QuotientHigh - OldCarry;
     RemainderSign;
F: if Geq Goto(Done);                    ! if remainder should be >= 0
     Remainder := Tos - Remainder;       ! set negative remainder
Done: Pop;                               ! restore stack
```

3.5.2 Unary

Unary special functions are special functions which require one argument to be specified (in the 'constant' phrase) along with the function keyword.

3.5.2.1 LeftShift

The LeftShift function is used to perform a logical left shift of the contents of the R register zero to 16 places.

The shifter control logic always keeps the latest shift control function loaded so that the shifter can shift a succession of words without respecifying the function.

3.5.2.2 RightShift

The RightShift function is used to perform a logical right shift of the contents of the R register zero to 16 places.

The shifter control logic always keeps the latest shift control function loaded so that the shifter can shift a succession of words without respecifying the function.

3.5.2.3 Rotate

The Rotate function is used to perform a logical rotate (right) of the contents of the R register zero to 16 places. Positive arguments from zero to 16 rotate R rightwards; negative arguments from 0 to -16 rotate R leftwards.

The shifter control logic always keeps the latest shift control function loaded so that the shifter can shift a succession of words without respecifying the function.

3.5.2.4 IOB

The IOB special function is used to supply an IO bus address. The high-order bit of the argument indicates the direction of transfer (IOB[7] = 1 for write, IOB[7] = 0 for read).

3.5.2.5 CntlRasterOp

The CntlRasterOp function is used to load an argument into the internal control register of the RasterOp hardware.

3.5.3 Binary

Binary functions are special functions which require two arguments. At this time, 'Field' is the only binary function implemented by the microassembler.

Field is used to extract a subfiled of R. The first argument is the bit index of the low order (right-most) bit of the field; the second argument is the width of the field.

CHAPTER 4

MICROASSEMBLER USER'S GUIDE

Section 4.1 describes the command level interface to the microassembler and placer, while section 4.2 describes the microassembler directives within a source file that are recognized by the microassembler.

4.1 MICROASSEMBLER COMMANDS

Before a microprogram can be run it must be assembled with PrqMic and then placed with PrqPlace. PrqMic translates the program into binary machine language, and PrqPlace assigns physical microstore locations to those instructions which are not assigned by the microprogrammer.

This section shows how to assemble and place a microprogram. A microprogram source file name has the form <src>.micro. <src> is called the "root name".

Once a microprogram has been assembled and placed, it can be used in one of the following ways:

1. Load it into another Perq, using ODTPerq.

2. Load it into the same Perq with the ControlStore module.

3. Write it into a boot file with MakeBoot.

4.1.1 Assemble

To assemble a microprogram, type "PrqMic" or "PrqMic <src>". If the root name is not supplied, PrqMic will prompt for it as follows:

"Root file name?"

4.1.2 Place

To place a microprogram, type

"PrqPlace"

   or

"PrqPlace <src> [<lst>]".

If <src> is not supplied, PrqMic will prompt for it. To place without a listing, type carraige return in response to the list filename prompt (if no arguments were supplied with the command line) or type the command line with <src> supplied but no <lst>. <lst> is the filename of the listing, if desired.

4.2 MICROASSEMBLER DIRECTIVES

There are several command lines which are directives to the microassembler and placer to perform special actions. These are indicated by the presence of a dollar sign ($) in column 1 of the command line. The entire line is considered to be a microassembler command, and thus, other microinstructions cannot be present on the same line.

4.2.1 INCLUDE

This directive inserts text from a file into the microprogram as though it were present in the original source file. The syntax is

   '$Include' filename

Included files cannot be nested, and so only the original source file may contain an include command.

4.2.2 TITLE

This directive prints a title string on the first line of every page of the assembly listing. The syntax is

   '$Title' TitleString

The first Title command sets the main title, which is printed at the left of each page. Each subsequent title command sets the subtitle which is printed at the right of each page.

4.2.3 NOLIST

This directive turns off listing. The syntax is

    '$NoList'

The listing is turned off until a List command (see 4.2.4) is
encountered. This command has an effect only if a listing has been
requested when the placer is executed (see section 4.1).

4.2.4 LIST

This directive resumes listing. The syntax is

    '$List'

The listing is resumed if it was turned off by a NoList (see 4.2.3)
command. This command has an effect only if a listing has been
requested when the placer is executed (see section 4.1).

4.2.5 PERQ1

The assembler will not recognize PERQ1A features. This is the
default. The syntax is

    $PERQ1

4.2.6 PERQ1A

The assembler recognizes PERQ1A features. $PERQ1 is the default. The
syntax is

    $PERQ1A

4.2.7 BASE

Registers with numbers less than 200 must be declared and used with a
percent sign (%) appended to their names. The percent sign indicates
that the register is one to which the base register is applied. This
directive is valid only in PERQ1A mode and is the default. The syntax
is

    $BASE

4.2.8 NOBASE

The percent sign is not required for registers with numbers less than

200.   This is used to maintain compatibility with the PERQ1. When
used, this option forces the programmer to be certain that the base
register contains 0.   This directive is valid only in PERQ1A mode.
The syntax is

   $NOBASE

CHAPTER 5

QUIRKS AND ODDITIES

The following quirks are known:

1. The Z field is inverted for shift functions (assembler fixes this).

2. The Op file is inverted on NextInst (assembler Opcode does it).

3. The Z field is inverted for all addresses (assembler fixes it).

4. IOB functions are executed twice if an abort occurs.

5. C19 will not be valid if an abort occurs on the test.

6. C19 test is inverted sense (i.e. jump if no carry out of bit 19).

7. Ustate 15:12 (the upper BMUX bits) are inverted.

8. Condition codes are not quite right after double precision adds and subtracts. See the 'Condition Codes' section (section 2.3.11).

9. The SF bits are inverted for JMP addresses (assembler fixes this).

10. Condition codes are invalid after reading MQ or Victim.

11. RBase must be loaded with inverted data.

12. Instructions containing the ALU operation $A + B + OldCarry$ or $A - B - OldCarry$ will not produce correct results if they are aborted.

13. Constant expressions may not be used as jump targets. The

assembler should allow this but does not.

14. On the PERQ1A, when the S register is used strictly as a
    12-bit counter by RepeatLoop, Repeat, or ThreeWayBranch, its
    upper 2 bits are ignored. In upper banks of the controlstore
    the assembler will generate leap jumps for LoadS and PushLoad
    instructions even though current bank jumps would be
    acceptable.

APPENDIX A

ERROR CODES

Error Index	Meaning
0	conflicting use of X field
1	conflicting use of Y field
2	conflicting use of A field
3	conflicting use of B field
4	conflicting use of W field
5	conflicting use of H field
6	conflicting use of ALU field
7	conflicting use of F field
8	conflicting use of SF field
9	conflicting use of Z field
10	conflicting use of CND field
11	conflicting use of JMP field
12	"8" or "9" in octal number
13	unknown symbol
14	undefined identifier
15	constant expected
16	identifier expected
17	"(" expected
18	")" expected
19	condition expected
20	identifer previously defined
21	label or constant expected
22	goto expected
23	bad shift count
24	bad field specification
25	"," expected
26	":=" expected
27	register expected
28	not allowed as left operand
29	not allowed as right operand
30	"Not" not allowed here
31	unexpected symbol
32	"OldCarry" expected

33	goto target not allowed
34	goto target expected
35	uncommented text found after ";"
36	missing "End;" supplied
37	condition required for this kind of jump
38	condition not allowed for this kind of jump
39	argument of OpCode larger than 377 (255)
40	address larger than 7777 (4095)
41	argument of CntlRasterOp larger than 377 (255)
42	argument of Iob larger than 377 (255)
43	XY register number larger than 377 (255)
44	"2".."9" in binary number
45	radix must be 2, 8, or 10 -- 8 assumed
46	number expected
47	number larger than 177777 (65536)
48	this pseudo-op must be on a line by itself
49	"Place" must be used only once, and must come first
50	first address greater than last address
51	size of program exceeds size allowed by "Place"
52	case number larger than 17 (15)
53	base address must have bits 2-5 equal to zero
54	unknown assembler option
55	"$Include" not allowed from an included file
56	"=" expected
57	missing ";" supplied
58	"$Perq1" or "$Perq1A" must come before any code
59	placement in more than one bank is not allowed
60	cross bank jump not allowed with this jump type
61	"Shift" not allowed with this jump type
62	conflicting result-bus specifications
63	address must have bits 9:2 equal to zero
64	interrupt number out of range 0..7
65	division by zero
66	"%" not allowed for register in the range 100..377 (64..255)
67	"%" required for registers in the range 0..77 (0..63)
68	not allowed as operand

APPENDIX B

WRITABLE CONTROL STORE (WCS) MAP

The following provides a map of the Writable Control Store (WCS) and describes the microcode register usage.

When you boot the system, VFY.MICRO runs memory diagnostics. EIOSYSB, CIOSYSB, or EIO5SYSB then loads microcode from the MBoot file. At boot time, microcode can only be loaded into WCS addresses 0 through 7377. At the end of the boot, only PERQ.MICRO and IO.MICRO are in the WCS. This leaves 7000 through 7777 for user-defined microcode. (You can use 7000 through 7377 for bootable special purpose microcode.)

PERQ.MICRO – PERQ Q-Code interpreter microcode. Temporary registers store state information during Q-Code interpretation. Registers: 3-21, 51-57, 64-67, 370. Temporary registers: 30-50, 61-63, 70-77. Placement: 0-4377.

IO.MICRO – Input/output microcode. Registers: 200-207, 211-217, 221-233, 235-252, 255-260, 262-266, 276, 277, 327, 373, 374. Temporary registers: 220, 261. Placement: 4400-5777.

LINK.MICRO – 16 bit parallel interface microcode (not normally booted into WCS). Registers: 350, 351. Placement: 6744-7400.

EIOSYSB, CIOSYSB, EIO5SYSB – system boot microcode. Registers: 0-74, 204, 267, 376. Placement: 7000-7777.

BOOT.MICRO – Boot microcode. Registers: 0-2, 4, 10, 20, 40, 100, 200, 252, 267, 277, 307, 330-340, 342-350, 357, 367, 373, 375, 376, 377. Placement: 0-777.

KRNL.MICRO – PERQ microcode kernel. Registers: 0, 356-377. Placement: 7400-7777.

GOODBY.MICRO – Power down microcode. Placement: 5000-5377.

ETHER10.MICRO – 10MBaud ETHERNET microcode. Registers: 300-321. Placement: Part of IO.MICRO.

RO.MICRO – Raster-op microcode. Registers: 100-124, 130-143, 370. Placement: Part of PERQ.MICRO.

LINE.MICRO – Line drawing microcode. Registers: 100, 101, 103-116. Placement: Part of PERQ.MICRO.

VFY.MICRO – Verifies that the hardware seems to work. Registers: 0-25, 300, 301, 370. Placement: 4000-6377.